



universidad
de león



Escuela de Ingenierías Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Sistemas de Información de Gestión y Business Intelligence

Memoria de la aplicación: LIVE

LIVE!

Javier Gómez Martínez
León, enero de 2022



ÍNDICE

1. Introducción	3
2. Descripción del problema	4
2.1. Contexto	4
2.2. Objeto	5
2.3. Variables observadas	5
2.3.1. Sobre la reproducción	5
2.3.2. Sobre el contexto	6
3. Herramientas y metodologías empleadas	7
3.1. Extracción de datos	7
3.2. Sistema de gestión de bases de datos	9
3.3. Back-end	9
3.4. Front-end	9
3.5. Otras herramientas de uso general	10
4. Aplicación desarrollada	11
4.1. Base de datos	11
4.2. Back-end	14
4.3. Front-end	16
4.3.1. Estructura general	16
4.3.2. Componentes	17
5. Algoritmos empleados	29
5.1. Modo automático de energía	29
5.2. Flujo de recomendación	33
5.2.1. ¿Existen peticiones?	34
5.2.2. Recopilación de información	34
5.2.3. Filtrado	35
5.2.4. Ordenación	38
6. Ejecución y análisis de resultados	41
6.1. Mismo track, misma hora, dos sesiones	41
6.1.1. Caso 1, 01:30 a 06:30	41
6.1.2. Caso 2, 23:30 a 05:30	43
6.2. Hoy no toca repetir	45
6.3. De género en género, del origen a la petición	48
7. Análisis DAFO	50
8. Evolución previsible del sistema	51
9. Lecciones aprendidas	52
10. Bibliografía, referencias y recursos	54



1. INTRODUCCIÓN

En la presente memoria se detalla la aplicación web *LIVE*, desarrollada para la asignatura Sistemas de Información de Gestión y Business Intelligence (SIBI), perteneciente al primer semestre del cuarto curso del Grado en Ingeniería Informática de la Universidad de León.

A lo largo de la misma, se detallará el problema abordado por la aplicación, las herramientas empleadas para su desarrollo, los algoritmos implementados así como su funcionamiento a través de varios casos de uso. De forma complementaria, se realiza un análisis DAFO (Debilidades, Amenazas, Fortalezas y Oportunidades) del proyecto y se proponen varias líneas de futuro para completar su funcionalidad.

Finalmente, se concluye haciendo un repaso de las distintas lecciones y experiencia adquirida por mi persona a lo largo de la elaboración de este proyecto.



2. DESCRIPCIÓN DEL PROBLEMA

2.1. CONTEXTO

La última década de los 2010 cambió radicalmente los hábitos en el consumo de música. Según el informe *Engaging with Music* de la Federación Internacional de la Industria Fonográfica (*IFPI*, por sus siglas en inglés), los entrevistados realizaban un 54% de su consumo musical en dichas plataformas. Y es que la irrupción de los servicios de streaming supuso dar acceso al gran público a millones de canciones a golpe de click. Este avance, de la mano de la reducción en los costes de promoción y autoproducción que permitió la informática personal, abrió una ventana al conocimiento de miles de artistas que hasta entonces dependían de los gustos de un departamento de *A&R* para ganarse una oportunidad en la industria musical.

Esta ventaja trajo rápidamente un problema que es inherente al ser humano: el de elección. La cantidad abrumadora de música hace difícil elegir qué canción poner. Toda esta situación se produce en medio de la explosión del *big data* y el resurgir de la inteligencia artificial, por lo que plataformas como la líder a la fecha de redacción de este documento, Spotify, deciden invertir en técnicas de análisis de audio y filtrado colaborativo, las cuales se basan en recomendar canciones que escuchan otros usuarios con un gusto similar. De esta manera, cada consumidor puede descubrir nueva música *de su cuerda*, sin salir de su zona de confort.

Pero no es éste el foco sobre el que se sitúa *LIVE*. Volviendo al ya mencionado estudio de *IFPI*, un 2% del consumo musical semanal se realiza en directo, es decir en conciertos de los propios artistas, o en el nicho de mercado de *LIVE*, los locales que viven de ofrecer una ambientación en torno a la música, como son los pubs o las discotecas.

Si bien en estos locales también se ofrece música pregrabada, como la que sale de unos auriculares al presionar *play* en una plataforma de streaming, existe una gran diferencia respecto al acto de escuchar música en la intimidad del salón de casa: la audiencia objetivo ya no es una única persona, sino cientos o incluso miles de ellas. Cada una tiene distintos gustos y situaciones personales, pero pese a ello, todas tienden a actuar como perfectos *animales sociales*, haciendo un disfrute colectivo.



Así mismo, dentro de las distintas aperturas, no existen dos sesiones iguales, ni dos públicos iguales. Incluso cabría afirmar que dentro de la misma sesión, no existen dos horas iguales, sino que cada una tiene sus peculiaridades: a unas, el público actuará de forma más enérgica, a otras, buscará canciones para moverse de forma más íntima.

En medio de esta situación, toda la responsabilidad recae en el DJ, convertido en el maestro de ceremonias que debe leer el estado del público, y saber complacerle adecuadamente. Y es aquí donde vuelve a aparecer el problema: ¿qué canción poner?

2.2. OBJETO

Por tanto, el objeto de este proyecto es el desarrollo de una aplicación web, que teniendo en cuenta una serie de datos sobre la música reproducida y el contexto de la sesión, emita recomendaciones sobre los siguientes *tracks* a reproducir.

2.3. VARIABLES OBSERVADAS

Como ya se ha mencionado en el anterior punto, podemos clasificar las variables observadas en dos grupos. Por un lado, aquellas relacionadas con la propia música reproducida, y por otro, las referentes al contexto que se da en un determinado momento de la sesión musical.

2.3.1. SOBRE LA REPRODUCCIÓN

Son aquellas inherentes a una determinada canción, las cuales ya son tenidas en cuenta por la mayoría de servicios de *streaming* para emitir recomendaciones. De todas ellas, LIVE tiene en cuenta las siguientes:

- **Tempo:** la velocidad a la que se ejecuta un determinado *track*, medida en pulsos por minuto (*BPM*, por sus siglas en inglés). Tiene especial importancia para el DJ, puesto que le será más fácil mezclar dos canciones con un tempo similar o próximo.
- **Energía:** entendida como la intensidad de una canción, medida en función de su *tempo*, volumen y ruido. Así, un determinado tema con una línea de bajo marcada y una velocidad alta tendrá más energía que una balada melódica.



- **Género:** clasificación a la que pertenece una determinada pieza musical, en función de características como su finalidad, patrón rítmico o instrumentación. Existen cientos de géneros musicales, y un determinado *track* podría no encajar a la perfección en un determinado género, o incluso estar clasificado en varios de ellos. En el *dataset* empleado por LIVE, se considera que un *track* pertenece a un solo género.
- **Relaciones:** proximidad de dos canciones, determinada por un valor similar de energía, pertenencia a un mismo género, o, especialmente, probabilidad de que un usuario escuche una después de la otra. En el *dataset* empleado por LIVE, no se asigna un peso o intensidad a estas relaciones.

2.3.2. SOBRE EL CONTEXTO

Son aquellas relacionadas con la situación que se está produciendo durante el transcurso de la sesión musical. LIVE considera las siguientes:

- **Energía del público:** entendida como la intensidad con la que el público responde a la música. Durante el transcurso de esta memoria, se entiende que esta variable está estrechamente correlacionada con la energía de una determinada canción, por lo que se hablará de una indistintamente de la otra.
- **Hora:** momento temporal en el que se produce la situación. En función de la duración de la sesión, un determinado momento estará más próximo al inicio o al final de la misma, lo que influye tanto en la energía del público como en la de los *tracks* que se desean reproducir.
- **Peticiones:** los *tracks* que el público pide en un determinado momento, y que, ya que atienden a gustos y necesidades personales, pueden estar más o menos relacionados con la reproducción de ese momento.
- **Historial:** las canciones que se han reproducido a lo largo de la sesión, y de las que, en la mayoría de los casos, no hay interés en volver a reproducir.



3. HERRAMIENTAS Y METODOLOGÍA EMPLEADAS

En esta sección se realiza una descripción de las distintas herramientas y metodología empleadas durante la elaboración del proyecto, desde la extracción del dataset utilizado como ejemplo o la implementación de la aplicación tanto en la parte correspondiente a la interfaz gráfica como en el *back-end*, hasta el sistema de gestión de bases de datos y otras herramientas de uso general.

3.1. EXTRACCIÓN DE DATOS

De acuerdo con las variables observadas descritas en el punto 2.3, en primer lugar se estudió cual podría ser la metodología más adecuada para su cuantificación y extracción, prestando especial atención a las relaciones entre canciones. Se valoró obtener los datos de forma manual, con una encuesta donde se ofrecieran una serie de canciones y el entrevistado tuviera que indicar, por respuesta espontánea, cuál sería la canción que reproduciría después de haber escuchado una determinada. Descartada esta opción dado lo costosa en tiempo y la cuestionable calidad de los datos que pudiera derivar (dado el sesgo y la subjetividad de cada individuo, únicamente mitigables con una cantidad de datos relativamente grande), se optó por obtenerlos de las plataformas de streaming. Analizadas las distintas APIs, se optó por recurrir a YouTube Music y Spotify, dividiéndose el proceso en dos fases diferenciadas.

- En primer lugar, para las relaciones entre las canciones, se optó por la plataforma musical de Google, ya que su API solo requiere de una autenticación mediante *cookie*, lo que ofrece la posibilidad de utilizar una *cookie* limpia, ofreciendo resultados no sujetos a los gustos y comportamiento previos de un usuario que haya utilizado YouTube con anterioridad.

YouTube Music ofrece una característica denominada *watchlists*, la cual ofrece hasta 25 canciones relacionadas con una determinada. Para construir el grafo de relaciones, se llevó a cabo un proceso de 3 iteraciones, donde se obtuvieron las 20 canciones relacionadas con una concreta. Como punto de partida, se utilizaron las 10 siguientes canciones, buscando que cada una de ellas representase un



estilo o género distinto, para obtener una base lo más representativa posible:

Artista	Título	Género
El Canto del Loco	Zapatillas	Pop-rock español
C. Tangana	Tú Me Dejaste de Querer	Flamenco/pop latino
Romeo Santos	Propuesta Indecente	Bachata
Farruko	Pepas	Reggaeton
Juan Magán	No Sigue Modas	Electro Latino
La Pegatina	Mari Carmen	Rumba
Ska-P	El Vals del Obrero	Ska/punk
Swedish House Mafia	Don't You Worry Child	EDM
Flo Rida	Good Feeling	Dance pop
Melendi	Caminando por la Vida	Rumba

En esta primera fase se utilizó fundamentalmente la **ytmusicAPI**, una API no oficial de YouTube Music implementada en Python, la cual emula solicitudes web de la misma manera que si éstas se realizaran navegando por el propio portal.

Obtenidos los datos, se elaboran 2 archivos CSV preliminares, uno con los títulos de las canciones y un identificador asociado, y otro con las relaciones, agrupadas en pares de canciones con sus respectivos identificadores. Dichos datos se incorporan a neo4j y se realiza una limpieza de ellos, identificando los componentes aislados y eliminándolos gracias a **NEuler - Graph Data Science Playground**.

- En una segunda fase, con las canciones resultantes, se obtienen sus valores asociados, como el tempo, la energía, su género, y otros datos de interés, como un enlace a su caratula o a una pequeña muestra de la canción. En este caso se recurre a la API de Spotify, realizando una búsqueda de las canciones y obteniendo su identificador en la plataforma sueca. Como efecto asociado, también se descartan una serie de *tracks* que sí estaban disponibles en YouTube pero no en otros servicios de *streaming*.



Para ello, se emplea la librería **spotify** de Python, autenticándose mediante los IDs asociados a una aplicación registrada en mi cuenta personal de Spotify.

3.2. SISTEMA DE GESTIÓN DE BASES DE DATOS

Como se ha mencionado anteriormente, el *DBMS* empleado es **neo4j**, un sistema de bases de datos no relacional, basado en grafos, completamente transaccional e implementado en Java. La utilización de un sistema como este permite aprovechar al máximo la propia naturaleza y características de la aplicación, basada en las relaciones de las distintas canciones entre sí y con el usuario. La base de datos se crea en local, usando la edición Desktop.

Cabe destacar también la utilización de dos de sus plugins, *Amazing Procedures on Cypher (APOC)* y la *Graph Data Science Library*, ambos desarrollados por la propia neo4j y que han permitido ejecutar algoritmos como *PageRank* o *Dijkstra* de forma nativa y con una sintaxis concisa.

3.3. BACK-END

Como nexo entre la base de datos y el *front-end*, se optó por **node.js**, y más concretamente, por su *framework* **Express**, dado que había hecho uso del mismo anteriormente en prácticas del Grado. Así, los accesos a las bases de datos se realizan a través de distintos *endpoints*, en forma de peticiones HTTP POST que son invocadas por el *front* cuando requiere del acceso o escritura de un determinado dato.

3.4. FRONT-END

Como estándar *de facto* en las aplicaciones web, se hizo uso de **JavaScript** y su *framework* **Vue.js**, especialmente indicado para páginas de una sola aplicación como LIVE. Otra razón importante en su elección es que éste es el *framework* utilizado mayoritariamente en trabajos anteriores de la asignatura, lo cual me permitió contar con un marco de referencia durante el proceso de desarrollo.

Además, se ha utilizado la librería **Vuetify** de Vue, la cual permite crear interfaces de usuario basadas en la norma **Material Design**



de Google, contando con una gran cantidad de componentes prediseñados y fácilmente personalizables a las características del diseño ideado para LIVE.

3.5. OTRAS HERRAMIENTAS DE USO GENERAL

A lo largo del proceso de desarrollo, se ha trabajado en el editor de código **Visual Studio Code**, dadas las extensiones existentes para el mismo, entre ellas **Vetur**, la cual ofrece ayuda y autocompleción para la sintaxis de Vue.js y JavaScript.

Para el diseño del logotipo de LIVE, se recurrió a **Photoshop**, aplicando una fuente Arial Bold Italic y un degradado púrpura.



4. APLICACIÓN DESARROLLADA

En el siguiente apartado se realiza una descripción del diseño de la aplicación. Como ya se intuye a partir del punto anterior, podemos dividir la estructura de la aplicación en 3 partes, la base de datos, la parte servidora o *back-end*, y la parte correspondiente a la interfaz de usuario o *front-end*.

4.1. BASE DE DATOS

La base de datos consiste en un grafo con dos tipos de nodos o vértices: personas (**Person**, en la base de LIVE) y canciones (**Song**). Hablamos por tanto de un grafo multipartito, aunque no existen nodos de tipo *Person* hasta que un usuario se registra en el servicio. En el *dataset* aportado, existen 3913 canciones, cada una de las cuales representa un nodo *Song*.

Los nodos **Song** cuentan a su vez con 10 propiedades diferenciadas: **id**, el cual es un identificador numérico único de la canción, generado por neo4j en la primera fase de extracción de datos detallada en el punto 3.1; **title**, que representa el título de la canción; **artist**, el artista que la interpreta; **idSpotify**, identificador alfanumérico único de la canción en el servicio Spotify (almacenado por si fuera necesaria una extracción de datos adicional); **genre**, género de la canción; **preview**, enlace a una muestra de 30 segundos en MP3 del tema; **cover**, enlace a la caratula de la canción; **energy**, energía de la canción, medida en un valor de 0 a 100; **bpm**, pulsos por minutos o tempo de la canción; y **date**, fecha de lanzamiento de la pieza.

Dentro de estos nodos *Song* es especial el caso del nodo deseo o **Desired Song** correspondiente a cada usuario, el cual almacena sus preferencias en cuanto a genero y energía. Así pues, cuenta con 3 propiedades: **energy**, valor deseado de energía; **genre**, que representa la elección en cuanto a mantener o no el género de la reproducción actual (posibles valores 'keep' y 'null'); y **title**, nombre del nodo, que siempre es una combinación de las cadena de caracteres 'Desired Song' más el nombre de usuario,

Los nodos *Person* cuentan con 7 propiedades: **email**, correo electrónico con el que se registra el usuario; **name**, nombre real o completo; **user**, usuario que identifica para autenticarse y **password**; **timeBegin**, momento temporal en el que el usuario ha



iniciado una sesión musical, medido en segundos desde el *epoch* (1 de enero de 1970); y **timeEnd**, momento en el que el usuario pretende finalizar la sesión, medido de igual manera en segundos. Estos dos últimos datos se toman al inicio de la sesión, como se detallará más adelante. El resto, se toman en el registro.

En cuanto a las relaciones entre estos nodos, destacan las de tipo **RELATED**, representando una relación entre dos canciones. Se pueden contabilizar 19203 relaciones de este tipo en el *dataset* de ejemplo, las cuales permiten formar una única componente conectada entre las canciones.

Otras relaciones de interés son **DESIRES**, que une una determinada persona con el nodo *Song* que almacena sus preferencias; **PLAYED**, que une un nodo *Person* con las canciones que ha reproducido a lo largo de una sesión; **LAST_PLAYED**, unión de un nodo *Person* con la última canción reproducida; y **WAS_ASKED**, unión entre un nodo *Person* con una canción que le ha sido solicitada. La relación **DESIRES**, junto a su correspondiente nodo de almacenamiento de preferencias *Desired Song*, son creados en el momento de registro y se mantienen mientras el usuario esté registrado. Las relaciones **PLAYED** se eliminan al cerrar sesión, mientras que solo existe una sola relación **LAST_PLAYED**, cuyo nodo destino se va actualizando a medida que se hace un cambio en la reproducción actual. Por último, la relación **WAS_ASKED** se mantiene mientras el usuario así lo desee.

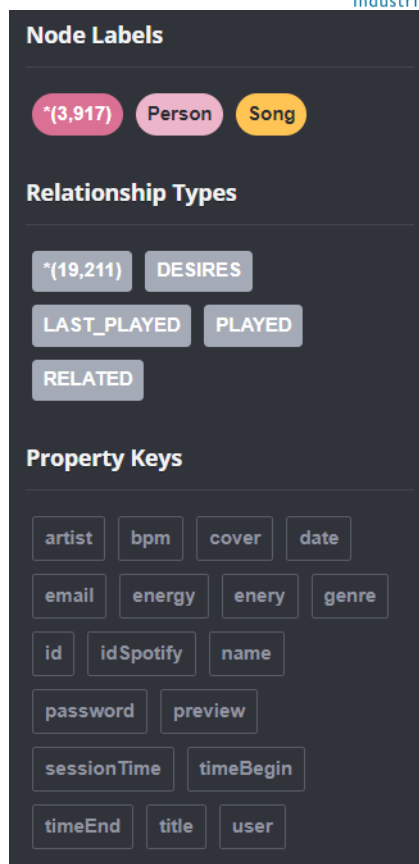


Figura 1: relación de nodos, relaciones y propiedades en neo4j

Ninguna de las relaciones descritas cuenta con propiedades o pesos de ningún tipo. Todas las relaciones son dirigidas, a excepción de las *RELATED*.

Para garantizar que no haya repetición ni de usuarios ni de canciones, es decir, que sean únicos, se crearon tres restricciones o *constraints*: ***EmailsUnique***, que garantiza que no existan dos usuarios con el mismo correo electrónico; ***UsernamesUnique***, para evitar que existan dos personas con el mismo nombre de usuario; y ***SongIdsUnique***, que garantiza la no existencia de dos canciones con el mismo identificador.

name	type	entityType	labelsOrTypes	properties
"EmailsUnique"	"UNIQUENESS"	"NODE"	["Person"]	["email"]
"SongIdsUnique"	"UNIQUENESS"	"NODE"	["Song"]	["id"]
"UsernamesUnique"	"UNIQUENESS"	"NODE"	["Person"]	["user"]



Figura 2: relación de restricciones en neo4j

Por último, es necesario destacar que algunas de los algoritmos empleados, como *PageRank* o *Dijkstra*, operan haciendo uso de lo que neo4j denomina grafo catalogado, una proyección de una determinada parte del grafo que se almacena en memoria principal y permite una ejecución más rápida de dichos algoritmos. En *LIVE*, el grafo catalogado empleado se denomina **grafoCanciones**, y es una proyección de los nodos *Song* sin propiedades junto a sus relaciones *RELATED*. Es necesario crear este grafo catalogado al arrancar la base de datos, ya que se borra al finalizar la instancia de la misma.

4.2. BACK-END

Como se detalló en el punto 3.3, el *back-end* está compuesto por distintos puntos de comunicación o *endpoints*, cada uno de los cuales ofrece acceso a una funcionalidad concreta:

- **/registro:** crea un nodo *Person* en la BBDD. Recibe como parámetros *nombre*, *email*, *usuario*, y *password*.
- **/crearNodoDeseo:** crea el nodo deseo o *Desired Song* asociado a un determinado usuario, donde se almacenarán sus preferencias en cuanto a género y energía. Este endpoint es solicitado después del registro, y recibe como parámetro el *usuario*.
- **/login:** comprueba que las credenciales de acceso sean correctas durante el proceso de inicio de sesión. Recibe como parámetros *usuario* y *password*.
- **/obtenerUltimaReproduccion:** recupera de la BBDD la última canción reproducida, si la hubiese, gracias a la relación *LAST_PLAYED*. Recibe como parámetro el *usuario*. Es solicitado después de completarse el inicio de sesión.
- **/eliminarUltimaReproduccion:** elimina la relación *LAST_PLAYED* de la BBDD. Recibe como parámetro el usuario. Es solicitado antes de actualizar la última reproducción.
- **/actualizarUltimaReproduccion:** almacena la *nueva* última reproducción, creando una relación *LAST_PLAYED*. Recibe como parámetros el *usuario* y el *idCancion*.
- **/obtenerPeticion:** recupera la petición de canción, si la hubiese, gracias a la relación *WAS_ASKED*. Recibe como parámetro el *usuario*. Es solicitado después de completarse el inicio de sesión.



- **/eliminarPetición:** elimina la relación *WAS_ASKED* de la BBDD. Recibe como parámetro el usuario. Es solicitado cuando el usuario desea descartar una petición, o cuando la actualiza.
- **/actualizarPetición:** almacena una petición de canción en la BBDD, creando una relación *WAS_ASKED*. Recibe como parámetros el *usuario* y el *idCanción*.
- **/establecerEnergia:** establece un valor de energía determinado en el nodo deseo asociado a un usuario. Es solicitado cuando se actualiza el valor de energía de forma manual. Recibe como parámetros *usuario* y *energía*.
- **/obtenerEnergia:** obtiene el valor de energía correspondiente a un determinado momento de la sesión. Para ello, recupera de la BBDD el valor de energía de la reproducción actual, así como los momentos de inicio y fin de la sesión, almacenados en el nodo *Person*. Es solicitado cuando se actualiza el valor de energía de forma automática. Recibe como parámetro el *usuario*. Una descripción más detallada del modo automático de energía se ofrece en el punto 5.
- **/establecerFechasInicioyFin:** permite configurar el inicio y fin de una determinada sesión, con el fin de utilizar el modo automático de energía. Es solicitado tras completarse el inicio de sesión, una vez el usuario proporciona los datos en el diálogo correspondiente. Recibe como parámetros *usuario*, *momentoInicio* y *momentoFin*.
- **/establecerGenero:** establece la preferencia en cuanto a mantener el género en las canciones a recomendar, la cual se almacena en el nodo deseo. Recibe dos parámetros, *usuario* y *genero*.
- **/obtenerGenero:** recupera la última preferencia en cuanto a género del nodo deseo. Es solicitado al completarse el inicio de sesión. Recibe como parámetro *usuario*.
- **/borrarHistorialReproduccion:** elimina todas las relaciones *PLAYED* asociadas a un determinado usuario, a excepción de la que conecta con la última reproducción. Es solicitado al cerrar sesión. Recibe como parámetro *usuario*.
- **/buscar:** busca todas las canciones en la BBDD cuyo título o artista contenga el conjunto de caracteres pasado como parámetro. Devuelve un máximo de 5 canciones que cumplan dicho criterio. Recibe como parámetro *termino*.



- **/obtenerRecomendaciones:** devuelve un máximo de 3 canciones recomendadas. Recibe como parámetro *usuario*. Una descripción detallada del flujo de recomendación se ofrece en el punto 5.

4.3. FRONT-END

4.3.1. ESTRUCTURA GENERAL

La estructura del *front-end* fue fácilmente generada gracias a la herramienta *CLI (Command Line Interface)* de Vue. En ella, distinguimos las siguientes partes:

- **Router:** permite crear la estructura de enrutamiento o enlaces de la aplicación. En LIVE, distinguimos tres rutas: la *raíz*, ocupada por la página de inicio de sesión o login; *register*, ocupada por la página de registro; y *dashboard*, página principal de la aplicación.
- **Store:** definida como un lugar de almacenamiento de *estados*, en LIVE es utilizada para almacenar información que es global a todo el proyecto, es decir, que es requerida por varios componentes. Destacamos dos estados: *usuario*, que hace referencia al nombre con el que una persona se autentica en la aplicación, y *setupFinalizado*, estado que toma como valor *true* cuando el usuario ha finalizado el inicio de sesión y además a configurado las fechas de inicio y fin de la sesión. A estos estados les acompañan sus correspondientes *getters*, *getUsuario* y *getSetup*, además de sus *setters* o *mutaciones*, *cambiarUsuario* y *setupFinalizado*.
- **Assets:** carpeta en la que se almacenan algunos recursos requeridos por la aplicación, como el logotipo de LIVE y el fondo presente en todas las vistas.
- **Plugins:** directorio en el que se almacenan algunas de las librerías utilizadas por las extensiones del proyecto, fundamentalmente *axios* y *Vuetify*.
- **Views:** son cada una de las páginas de la aplicación, de manera que hay una correspondencia 1 a 1 entre cada una de ellas y las rutas definidas en el *router*. Aunque existe la posibilidad de escribir el código de las mismas directamente, en el diseño escogido se opta por delegar esta tarea en los componentes. En definitiva, contamos con 3 vistas: *Login*, que encapsula el componente



SignIn; *Register*, que encapsula el componente *Registro*; y *Dashboard*, que encapsula el componente *HomeDashboard*.

- **Componentes:** abarcan todo el código de creación propia de la aplicación. Distinguimos 3 componentes padre, los ya descritos *SignIn*, *Register*, y *HomeDashboard*. Este último, requiere a su vez de tres componentes hijo: *ReproduccionActualDashboard*, *ParametrosDashboard* y *RecomendacionesDashboard*.

4.3.2. COMPONENTES

- **SignIn:** es el componente que implementa la página de inicio de sesión. Su diseño básico está formado por una tarjeta (v-card de Vuetify), la cual contiene dos campos para el usuario y la contraseña, así como un botón de inicio de sesión, que invoca el método *login()*. Este método, como muchos otros detallados a continuación, realiza una petición *HTTP POST* a su *homólogo* del back-end con los parámetros detallados en el punto 4.2. Finalmente, contamos con un enlace a la página de registro, y un método *limpiarFormulario()*, invocado cuando las credenciales introducidas no son correctas o se produce algún tipo de error.

En este componente, y en prácticamente todos, se hace uso de una *snackbar*, una barra informativa situada en la parte inferior de la pantalla, que emerge si se ha producido algún tipo de error. Su código es idéntico en todos los componentes.

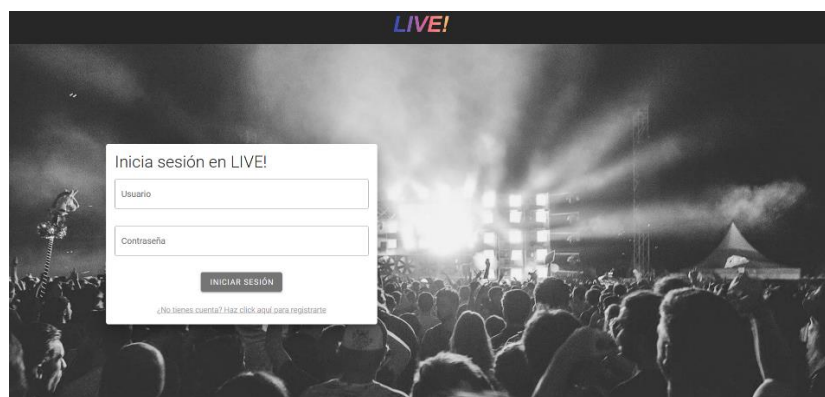


Figura 3: vista Login, formada por el componente *SignIn*

```
login(){
  if(this.usuario.length == '0' || this.password.length == '0'){
    this.textoSnackBar = "Por favor, rellena todos los campos";
    this.snackbar = true;
  }else{
    axios
      .post('http://localhost:3000/login' ,{
        usuario: this.usuario,
        password: this.password
      })
      .then((response) => {
        if(JSON.stringify(response.data) == JSON.stringify({msg: 'Incorrecto'})){
          this.textoSnackBar = "Usuario o contraseña incorrectos";
          this.snackbar = true;
          this.limpiarFormulario();
        }else if(JSON.stringify(response.data) == JSON.stringify({msg: 'Error'})){
          this.textoSnackBar = "Se produjo un error. Inténtalo de nuevo";
          this.snackbar = true;
          this.limpiarFormulario();
        }else{
          this.cambiarUsuario(this.usuario);
          this.$router.push({name: 'Dashboard'});
          this.limpiarFormulario();
        }
      })
      .catch(error =>{
        this.textoSnackBar = "Se produjo un error. Revisa los datos e inténtalo de nuevo";
        this.snackbar = true;
        this.limpiarFormulario();
        console.log(error);
      })
  }
},
limpiarFormulario(){
  this.usuario = "";
  this.password = "";
},
```

Figura 4: código de los metodos login() y limpiarFormulario

```
<!-- SnackBar advertencia -->
<v-snackbar v-model="snackbar" class="d-flex">
  {{ textoSnackBar }}
  <v-btn color="white" fab icon @click="snackbar = false">
    <v-icon>mdi-close</v-icon>
  </v-btn>
</v-snackbar>
```

Figura 5: código de la snackbar

```
<!-- Card de inicio de sesión -->
<v-card elevation="15" width="500">
  <v-card-title>
    <h2 class="font-weight-light">Inicia sesión en LIVE!</h2>
  </v-card-title>
  <v-card-text>
    <v-form @enter="login" @submit.prevent="login">
      <v-text-field
        label="Usuario"
        outlined
        clearable
        v-model="usuario"
      ></v-text-field>
      <v-text-field
        label="Contraseña"
        outlined
        clearable
        v-model="password"
        type="password"
      ></v-text-field>
      <v-btn class="font-weight-regular" color="grey darken-1 white--text" type="submit">Iniciar sesión</v-btn>
    </v-form>
  </v-card-text>
</v-card>
```

Figura 6: código de la tarjeta de inicio de sesión



- **Registro:** componente que implementa la página de registro. No existen grandes diferencias entre éste y el componente de inicio de sesión. Al igual que el anterior, cuenta con una tarjeta con cuatro campos para introducir nombre, email, usuario y contraseña, y unos métodos para enviar la solicitud de registro y limpiar el formulario, **registro()** y **limpiarFormulario()**. Cuenta de igual forma con la *snackbar* informativa.

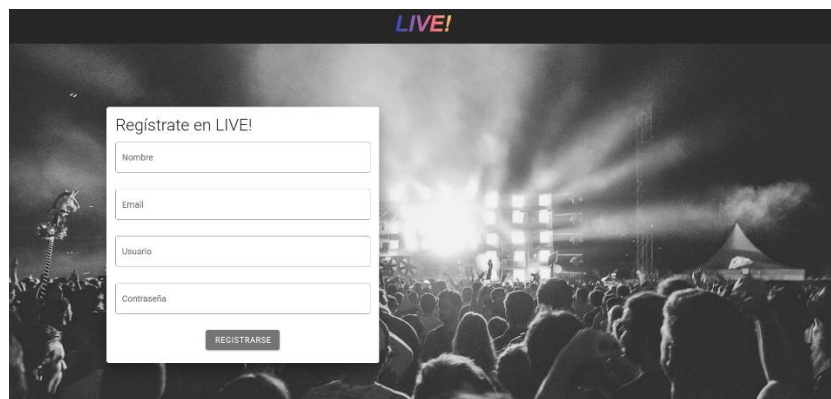


Figura 7: vista Register, formada por el componente Registro

- **Dashboard:** página principal de la aplicación. Aparte de contar con 3 componentes hijos, **ParámetrosDashboard**, **RecomendacionesDashboard**, y **ReproduccionActualDashboard**, en los que delega la práctica totalidad de su funcionalidad, cuenta con un cuadro de diálogo desplegado al iniciar sesión, en el cual se pueden configurar las fechas de inicio y fin de la sesión, necesarias para que funcione el modo automático de energía. Este diálogo es un *v-dialog* de Vuetify, el cual contiene una tarjeta como las detalladas anteriormente, con los campos de fecha y hora de inicio y fin. Para seleccionar estas fechas, se ofrece un calendario desplegable, *v-date-picker*. Para la hora, se ofrece un desplegable similar, *v-time-picker*. Una vez se pulsa el botón aceptar, se llama al método **establecerFechasInicioYFin()**, que comprueba la validez de los datos introducidos y llama a su homólogo del *back-end*. Una vez obtiene respuesta, llama a la mutación **finalizarSetup()**, ubicada en la *Store*, para indicar que el proceso de configuración inicial ha finalizado. Por último, solicita que se calcule el valor de

energía utilizando el modo automático, una funcionalidad que reside en el componente *ParametrosDashboard*, referenciado simplemente como *parámetros*.

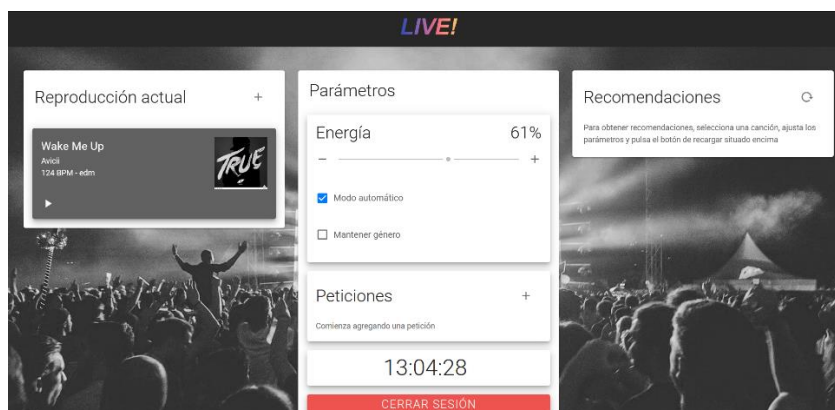


Figura 8: Dashboard compuesto por sus tres componentes hijo

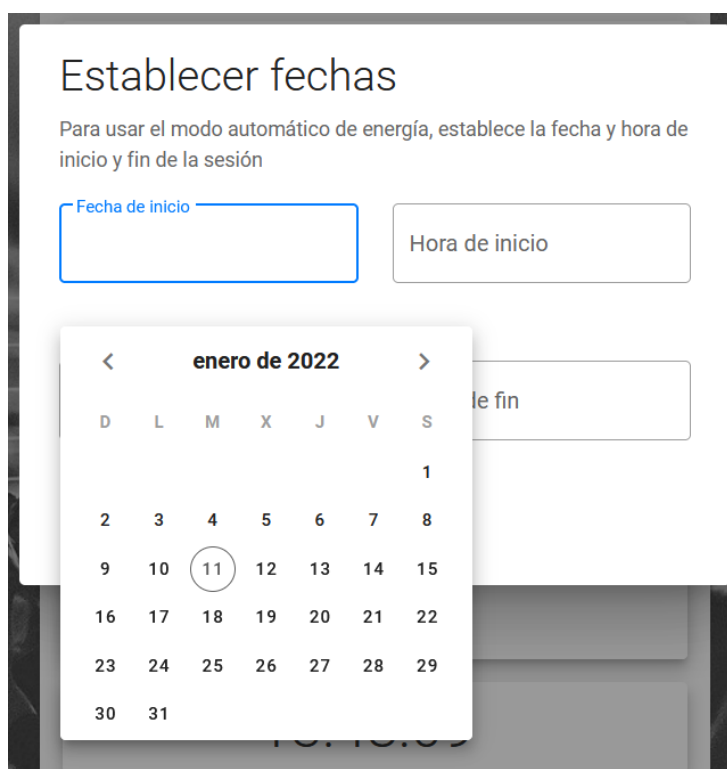


Figura 9: selector de fechas con calendario desplegable



```
establecerFechasInicioYFin(){
    var momentoActual = Date.now();
    var momentoInicio = new Date((this.fechaInicio + " ") + (this.horaInicio + ":00")).getTime();
    var momentoFin = new Date((this.fechaFin + " ") + (this.horaFin + ":00")).getTime();

    if(momentoActual < momentoInicio || momentoActual > momentoFin || this.fechaInicio == null || this.fechaFin == null || this.horaInicio == null || this.horaFin == null){
        this.textoSnackbar = "La fechas introducidas no son válidas. Revisa los datos introducidos y la hora actual";
        this.snackbar = true;
    }else{
        axios
            .post('http://localhost:3000/establecerFechasInicioYFin', {
                usuario: this.getUsuario,
                momentoInicio: momentoInicio,
                momentoFin: momentoFin
            })
            .then(response => {
                if(JSON.stringify(response.data) == JSON.stringify({msg: 'Error'})){
                    this.textoSnackbar = "Ocurrió un error al establecer las fechas. Vuelve a intentarlo";
                    this.snackbar = true;
                }else{
                    this.dialogoInicial = false;
                    this.finalizarSetup();
                    this.$refs.parametros.actualizarEnergia;
                }
            })
            .catch(error => {
                this.textoSnackbar = "Ocurrió un error al establecer las fechas. Vuelve a intentarlo";
                this.snackbar = true;
                console.log(error);
            })
    }
}
```

Figura 10: método establecerFechasInicioYFin()

- **ReproduccionActualDashboard:** muestra la canción seleccionada en ese momento, desplegando el título de la canción, su artista, valor de BPM, y género. Es por tanto la principal representación de las variables relacionadas con la reproducción, detalladas en el punto 2.3.1. Dicha representación se realiza de nuevo sobre una *v-card*, que también ofrece un botón de *play* para escuchar una muestra de la canción de 30 segundos, así como una imagen con la caratula del sencillo. Encima, distinguimos un botón +, el cual pulsado, ofrece un buscador para poder seleccionar otra canción. Dicho buscador es un *v-dialog*, que ofrece un cuadro de búsqueda y hasta 5 sugerencias, organizadas en una *v-list*. En cuanto a los métodos de este componente, distinguimos principalmente ***alterarReproduccion()***, invocado para reproducir o pausar la canción seleccionada; ***seleccionarTrack()***, invocado al cambiar de canción; ***activarDialogo()***, para desplegar el buscador; y otros métodos que invocan a sus homólogos en el *back-end*, como son ***buscar()***, ***alterarReproduccion()***, ***recuperarReproduccion()*** y ***actualizarUltimaReproduccion()***.

```
activarDialogo(){
    this.dialog = true;

    if(this.botonPlay == true){
        this.alterarReproduccion();
    }
}
```

Figura 11: código de activarDialogo()

```
seleccionarTrack(item, firstTime){
    this.cancionSeleccionada = item;
    this.dialog = false;
    this.isCancionSeleccionada = true;

    if(this.cancionSeleccionada.preview != 'null'){
        this.audio = new Audio(this.cancionSeleccionada.preview);
    }

    if(firstTime == false){
        this.actualizarUltimaReproduccion();
    }
},
```

Figura 12: código de seleccionarTrack()

```
alterarReproduccion(){
    if(this.cancionSeleccionada.preview == 'null'){
        this.textoSnackbar = 'Lo sentimos, no hay disponible una muestra de esta canción';
        this.snackbar = true;
    }else{
        this.botonPlay = !this.botonPlay;

        if(this.botonPlay == false){
            this.audio.pause();
        }else{
            this.audio.play();
        }
    }
},
```

Figura 13: código de alterarReproduccion()

```
// Dialogo de búsqueda ->
<v-dialog v-model="dialog" width="500">
  <v-card class="pb-1 pt-1 pl-1 pr-1">
    <v-btn @click="dialog = false" fab icon class="ml-2 mb-2 mt-2"><v-icon>mdi-close</v-icon></v-btn>
    <h1 class="font-weight-light pl-7">Buscar</h1>

    <v-card-text>
      <div>
        <!-- Cuadro de búsqueda -->
        <v-text-field
          label="Introducir título o artista"
          outlined
          clearable
          v-model="busqueda"
        ></v-text-field>

        <!-- Items sugeridos -->
        <v-list>
          <v-list-item link :disabled="item.titulo == undefined" @click="seleccionarTrack(item, false)" v-for="(item,index) in resultadosBusq">
            <v-list-item-content>
              <v-list-item-title>{{item.titulo}}</v-list-item-title>
              <v-list-item-subtitle>{{item.artista}}</v-list-item-subtitle>
            </v-list-item-content>
          </v-list-item>
        </v-list>
      </v-card-text>
    </v-card>
  </v-dialog>
```

Figura 14: código del diálogo de búsqueda



```
<!-- Tarjeta de canción seleccionada -->
<v-col v-else>
  <v-card dark color="grey darken-2" elevation="7">
    <div class="d-flex flex-no-wrap justify-space-between">
      <div>
        <v-card-title class="font-weight-regular white--text">
          {{cancionSeleccionada.titulo}}
        </v-card-title>
        <v-card-subtitle>
          {{cancionSeleccionada.artista}}
          <br>
          {{cancionSeleccionada.bpm}} BPM - {{cancionSeleccionada.genero}}
        </v-card-subtitle>
        <v-btn fab icon @click="alterarReproduccion()">
          <v-icon>{{botonPlay ? 'mdi-pause' : 'mdi-play'}}</v-icon>
        </v-btn>
      </div>
      <div>
        <v-avatar class="mt-4 mr-4" size="95" tile>
          <v-img :src="cancionSeleccionada.cover"></v-img>
        </v-avatar>
      </div>
    </div>
  </v-card>
</v-col>
```

Figura 15: código de la tarjeta de reproducción actual



Buscar

Introducir título o artista

el canto

X

Insoportable
El Canto del Loco

La Madre de Jose
El Canto del Loco

Aunque Tu No Lo Sepas
El Canto del Loco

Quiero Aprender de Ti
El Canto del Loco

Puede Ser
El Canto del Loco

Figura 16: diálogo de búsqueda

Reproducción actual

+

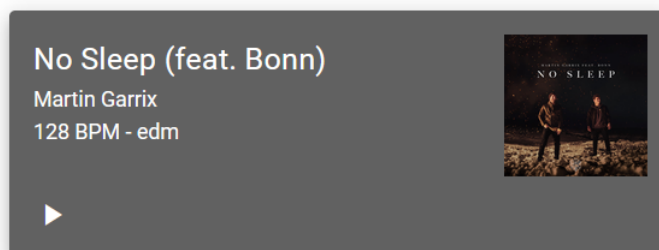


Figura 17: representación de ReproduccionActual

- **ParametrosDashboard:** es la principal representación de las variables de contexto descritas en el punto 2.3.2. El componente está formado por 3 tarjetas: una de **energía**, la cual se puede ajustar de forma automática o manual, gracias a un *v-slider*, pudiendo también en esta tarjeta seleccionar si se desea mantener el género de la reproducción actual en las recomendaciones; otra de **peticiones**, la cual en esta primera versión permite seleccionar una petición, y que ofrece un buscador idéntico al del componente de reproducción actual; un **reloj** en tiempo real; y finalmente, el botón de **cerrar sesión**. Dentro de los métodos, encontramos *seleccionarTrack()*, prácticamente idéntico al método homónimo del componente de reproducción actual; *deseleccionarTrack()*, para descartar una petición; *cambiarModo()*, accionado cuando activamos o desactivamos el modo automático de energía; y otros como *buscar()*, *recuperarPeticion()*, *actualizarPeticion()*, *actualizarEnergia()*, *recuperarGenero()*, *cerrarSesion()*, *actualizarGenero()* o *eliminarPeticion()*, que invocan sus homólogos en el *back-end*. Por último, es destacable la presencia de dos *intervalos*, los cuales ejecutan código cada x tiempo. Agrupados bajo la etiqueta *created*, el primero de ellos tiene como finalidad actualizar la hora cada segundo, y el segundo, actualizar la energía en modo automático cada 5 segundos.


```
cambiarModo(){
    this.actualizarEnergia;
}
```

Figura 18: método `cambiarModo()`

```
deseleccionarTrack(){
    this.isPeticiónSeleccionada = false;
    this.petición = null;

    this.eliminarPetición();
},
```

Figura 19: método `deseleccionarTrack()`

```
<!-- Tarjeta de energía -->
<v-card class="mb-4" elevation="7">
  <v-card-title>
    <h2 class="font-weight-light">Energía</h2>
    <v-spacer></v-spacer>
    <h2 class="font-weight-light">{{porcentajeEnergia}}%</h2>
  </v-card-title>
  <v-card-text>
    <!--Slider-->
    <v-slider :disabled="checkboxMode" v-model="porcentajeEnergia" track-color="grey" min="0" max="100">
      <template v-slot:prepend>
        <v-icon color="grey darken-2">mdi-minus</v-icon>
      </template>

      <template v-slot:append>
        <v-icon color="grey darken-2">mdi-plus</v-icon>
      </template>
    </v-slider>
    <v-checkbox
      @change="cambiarModo()"
      v-model="checkboxMode"
      label="Modo automático"
    ></v-checkbox>
    <v-checkbox
      :disabled="!checkboxGeneroActivado"
      v-model="checkboxGenero"
      label="Mantener género"
      @change="actualizarGenero"
    ></v-checkbox>
  </v-card-text>
</v-card>
```

Figura 20: código de la tarjeta de energía

```
<!-- Tarjeta de hora -->
<v-card class="mb-4" elevation="7">
  <v-card-title class="justify-center mr-3">
    <h1 class="font-weight-light">{{time}}</h1>
  </v-card-title>
</v-card>

<!-- Cerrar sesión -->
<v-btn @click="cerrarSesion" class="red lighten-1" block elevation="7">
  <h2 class="font-weight-light white--text">Cerrar sesión</h2>
</v-btn>
```

Figura 21: código de la tarjeta de hora y el botón “cerrar sesión”



```
<!-- Tarjeta de peticiones -->
<v-card class="mb-4" elevation="7">
  <v-card-title>
    <h2 class="font-weight-light">Peticiones</h2>
    <v-spacer></v-spacer>
    <v-btn @click="dialog = true" fab icon><v-icon>mdi-plus</v-icon></v-btn>
  </v-card-title>
  <v-card-text v-if="!isPeticiónSeleccionada">
    Comienza agregando una petición
  </v-card-text>

  <v-list v-else>
    <v-list-item @click="deseleccionarTrack">
      <v-list-item-content>
        <v-list-item-title>{{petición.título}}</v-list-item-title>
        <v-list-item-subtitle>{{petición.artista}}</v-list-item-subtitle>
      </v-list-item-content>
    </v-list-item>
  </v-list>
</v-card>
```

Figura 22: código de la tarjeta de peticiones

```
this.interval = setInterval(() => {
  /*Formato de hora*/
  this.time = Intl.DateTimeFormat(navigator.language, {
    hour: 'numeric',
    minute: 'numeric',
    second: 'numeric'
  }).format();
}, 1000);

/*Actualizar la energía en modo automático cada 5000 ms (5 segundos)*/
this.intervalBis = setInterval(() => {
  if(this.checkboxMode == true && this.getSetup == true){
    axios
      .post('http://localhost:3000/obtenerEnergia', {
        usuario: this.getUsuario
      })
      .then(response => {
        if(JSON.stringify(response.data) == JSON.stringify({msg: 'Error'})){
          this.textoSnackbar = "Ocurrió un error al actualizar la energía";
          this.snackbar = true;
        }else if(JSON.stringify(response.data) == JSON.stringify({msg: 'Pendiente'})){
          this.textoSnackbar = "Selecciona una canción para poder obtener la energía en modo automático";
          this.snackbar = true;
        }else if(JSON.stringify(response.data) == JSON.stringify({msg: 'Finalizado'})){
          this.textoSnackbar = "El tiempo de sesión ha finalizado. Desactiva el modo automatico";
          this.snackbar = true;
        }else{
          this.porcentajeEnergia = parseInt(response.data.value);
        }
      })
      .catch(error => {
        this.textoSnackbar = "Ocurrió un error al actualizar la energía";
        this.snackbar = true;
        console.log(error);
      })
  }
}, 5000);
}
```

Figura 23: código de los intervalos

Parámetros

Energía46%

-

+

☐ Modo automático

☐ Mantener género

Peticiones

+

Don't Look Down (feat. Usher)
Martin Garrix

15:29:09

CERRAR SESIÓN



Figura 24: representación del componente
ParametrosDashboard

- **RecomendacionesDashboard:** componente que muestra las recomendaciones, una vez las ha obtenido de la base de datos basándose en las variables de la reproducción actual y las de contexto. Para obtener recomendaciones, cuenta con un botón de recarga en su parte superior, el cual invoca el método ***obtenerRecomendaciones()***, que a su vez llama a su homólogo del *back-end*. El otro método con el que cuenta este componente es ***alterarReproduccion()***, que, al igual que el método homónimo en la reproducción actual, es el encargado de pausar o reproducir las canciones recomendadas. Se muestra siempre un máximo de 3 recomendaciones, distinguidas en los colores, *indigo lighten-1*, *deep-orange lighten-2* y *orange lighten-2*, nativos de Vuetify y coherentes con la identidad visual de LIVE.



Recomendaciones





Bella ciao (Hugel Remix)
El Profesor
126 BPM - house



Hear Me Now (feat. Zeeba)
Alok
121 BPM - electro house



High On Life (feat. Bonn)
Martin Garrix
128 BPM - edm



*Figura 25: representación del componente
RecomendacionesDashboard*



5. ALGORITMOS EMPLEADOS

En el presente apartado se realiza una descripción de los algoritmos empleados por LIVE para calcular los valores necesarios para apoyar o cumplir con su funcionalidad: realizar una recomendación musical basada en contexto.

5.1. MODO AUTOMÁTICO DE ENERGÍA

Tal como se describe en el apartado 2.3.2, entre las variables que determinan el contexto de un determinado momento de la sesión musical encontramos el historial de reproducción, las peticiones, la hora y la de energía

Entendida tanto como la intensidad de la música como la de la respuesta del público a la misma, desde el inicio del desarrollo de LIVE se pensó en plantear tanto la opción de que el DJ pudiese regularla de forma manual, dando el *ritmo* que él deseara a la sesión, como que ésta pudiera ser calculada de forma automática, en consonancia con otras variables, garantizando al usuario una sesión musical completamente armónica.

Así pues, el primer paso fue definir qué se entiende por armonía. En LIVE, una sesión armónica es ésa que cumple un ciclo, acabando igual que empezó. Por tanto, debe tener un momento de máxima intensidad, en el que la gente ya no tema dar todo de sí misma, y otros más tranquilos, bien para no quemar los grandes temas de la noche antes de que el local esté siquiera lleno, o para que el público intuya cuando la sesión esta tocando su fin.

Un ciclo en definitiva, que tenga una forma de parábola. Tomándose una función donde t sea el momento de la noche e y la energía, siendo $t=0$ el momento inicial, y $t=1$ el momento final, tratando de que la y toque su máximo valor en $t=0.5$.

Si observamos el *dataset* de ejemplo, con la energía medida en una escala de 0 a 100, podemos comprobar como sólo 64 de los 3913 nodos *Song* iniciales tienen un valor de energía menor a 30, mientras que 13 tienen un valor menor a 20. Podemos afirmar con casi total seguridad que estas trece canciones son una rareza y tienen un valor de energía extremadamente bajo, lo que también

debería evitarse si no se desea que el público se aburra y abandone el local. Otro detalle a tener en cuenta es la procedencia del dataset: Spotify, la cual ha obtenido los valores de energía mediante el análisis de una parte concreta de la canción, y no de su totalidad. Cabe así posibilidad de que un determinado valor de energía del dataset no sea lo suficientemente representativo.

Descartando los *tracks* con una energía sorprendentemente baja, y planteando un margen de seguridad de $\pm 10\%$, podemos proponer una función donde el valor de energía, o y , para $t = 0$ y $t = 1$ sea 30, mientras que el valor de y en $t = 0.5$ sea 90.

Dicha función es la siguiente:

$$y_i = 240t - 240t^2 + 30$$

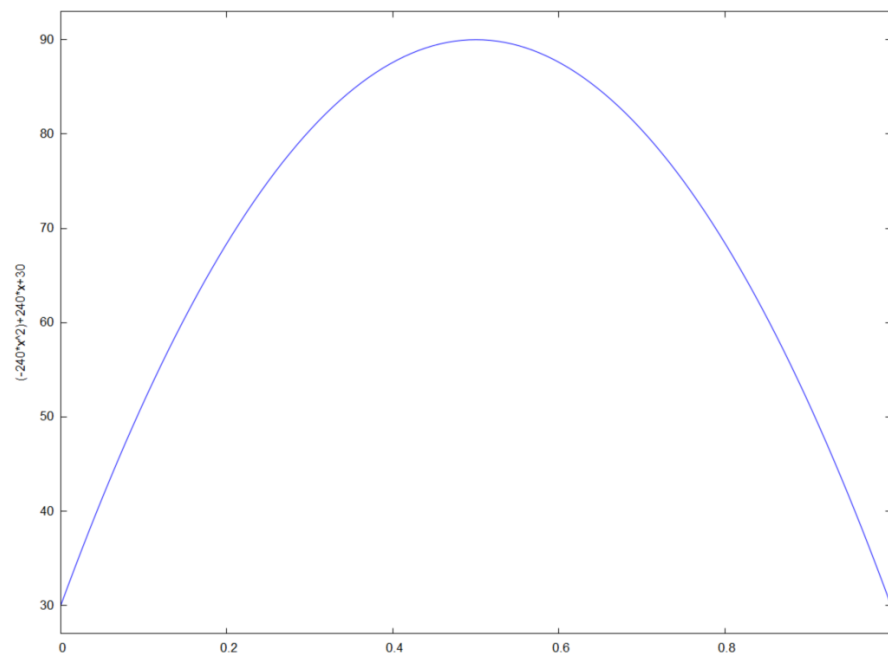


Figura 26: gráfica de la función de energía ideal automática

El momento de la sesión es convertido al intervalo (0,1) mediante la aplicación de la fórmula resultante de resolver el siguiente sistema:



$$at_i + b = 0$$

$$at_f + b = 1$$

Donde a y b son las dos constantes a despejar, y t_f y t_i , los momentos final e inicial, medidos en la unidad correspondiente, en el caso de LIVE, segundos desde el *epoch*.

La función enunciada ilustrada en la *Figura 26* debe entenderse como la **energía ideal** (y_i), es decir, como el valor de energía que se desearía generar en el público para garantizar la armonía anteriormente mencionada.

Pero desear algo no significa que realmente vaya a suceder, por mucho que uno ponga todo su empeño. Por lo que para ofrecer recomendaciones realistas, es necesario ajustar el valor de energía para acercarlo a la realidad. Una vía sería que dicho ajuste fuese realizado de forma manual, en cuyo caso el modo automático perdería todo su sentido, pudiéndose realizar desde el principio todo el ajuste de energía de forma manual sin gastar recursos de la máquina cada 5 segundos (intervalo con el que se actualiza la energía, como se explicó en el punto 4.3).

La otra vía es emplear el resto de los datos contextuales. Si para calcular la energía ideal se utilizó como soporte la hora, para aproximar la energía a su valor real utilizaremos la reproducción actual, que, elegida siempre en último lugar por el DJ, nos puede dar una imagen más realista del contexto de la sesión.

Se puede proponer así calcular la energía real como la media entre la energía ideal y el valor de energía de la reproducción actual:

$$y_r = \frac{(y_a + y_i)}{2}$$

Siendo y_i la energía ideal, e y_a el valor de energía de la reproducción actual.

Ésta es la fórmula empleada para el cálculo de la energía en modo automático, y puede observarse en el código del *endpoint obtenerEnergia()* del *back-end*. En primer lugar, se recupera de la base de datos el momento inicial y final de la sesión, así como la energía de la reproducción actual. En segundo lugar, se crea un nuevo objeto de la clase *Date*, obteniendo el momento actual. Después, se resuelve el sistema antes mencionado para convertir el momento actual a un valor comprendido entre (0,1) (haciendo uso de la librería *Nerdamer*), y finalmente, aplicando la función de energía real y respondiendo al *front-end*, además de escribiendo el valor actualizado al *nodo deseo* de la BBDD.

```
var momentoActual = new Date().getTime();
var energiaReproduccionActual = result.records[0]._fields[0];
var momentoInicio = result.records[0]._fields[1];
var momentoFin = result.records[0]._fields[2];

if(momentoActual > momentoFin){
  res.json({msg: 'Finalizado'});
}else{
  var ecuacion = nerdamer.solveEquations([(momentoInicio + 'a + b = 0'),(momentoFin + 'a + b = 1')]).toString();
  var a = parseFloat(ecuacion.split(',')[1]);
  var b = parseFloat(ecuacion.split(',')[3]);

  var momentoNormalizado = momentoActual*a + b;
  energy = {value: (((240*momentoNormalizado) - (240*momentoNormalizado*momentoNormalizado) + 30) + energiaReproduccionActual)/2};

  const anotherSession = driver.session();
  var secondQuery = "MATCH (p:Person {user:'" + usuario + "'})-[:DESIRE]->(s:Song) SET s.energy=" + energy.value;

  const secondResultPromise = anotherSession.run(secondQuery);
  secondResultPromise
    .catch(error => {
      res.json({msg: 'Error'});
      console.log(error);
    })
    .then(() => session.close());

  res.send(energy);
}
```

Figura 27: código de cálculo de la energía en *obtenerEnergia()*

5.2. FLUJO DE RECOMENDACIÓN

Cuando el usuario considere que ha ajustado correctamente las preferencias de contexto, y haya seleccionado una canción como reproducción actual, puede obtener recomendaciones pulsando el botón de refresco del componente de recomendaciones. En ese momento, se inicia un proceso en el que se recuperan una serie de valores de la base de datos, y en función de ellos, sigue un itinerario u otro.

Cada una de las fases detalladas a continuación viene condicionada no solo por dichos valores, sino también por los resultados obtenidos en cada fase. Si en algún momento determinado del proceso se obtienen 3 resultados o menos, se devuelven inmediatamente al *front-end* y no se continúa hacia las fases sucesivas.



1. ¿EXISTEN PETICIONES?

La primera consulta a la BBDD comprueba si hay alguna petición en ese momento, gracias a la relación *WAS_ASKED*. Concretamente, se realiza la siguiente *query*:

```
/*Fase 1: Comprobación de si existen peticiones*/  
var firstPhaseQuery = "MATCH (p:Person)-[:WAS_ASKED]->(s:Song) WHERE p.user='" + usuario + "' RETURN s";
```

Figura 28: comprobación de existencia de peticiones

La petición puede devolver el nodo *s*, y solo un nodo *s* (precondición sobre la relación *WAS_ASKED* expuesta en el punto 4.1), o no devolver nada. Se obtienen así dos posibles itinerarios, que marcarán el resto del proceso.

2. RECOPIACIÓN DE INFORMACIÓN

a) NO EXISTEN PETICIONES

En este caso, se recupera diversa información del nodo de reproducción actual, incluyendo **id**, **género**, **energía** y **bpm**. Se realiza un proceso similar con el *nodo deseo*, recuperando la preferencia en cuanto al **género** (posibles valores, 'keep' y 'null') y **energía**, ya se haya calculado ésta de forma automática o manual, lo cual es irrelevante para el resto del flujo.

```
/*Fase 2: obtención de información sobre la recomendación actual y los parámetros*/  
var secondPhaseQuery = "MATCH (s:Song)-[:LAST_PLAYED]-(p:Person {user:'" + usuario + "'})-[:DESIRES]->(d:Song) RETURN s.id, toFloat(s.bpm), s.genre, toFloat(d.energy), d.genre";
```

Figura 29: recopilación de información en el paso 2.a

b) EXISTEN PETICIONES

Se recupera toda la información expuesta en el paso 2.a, a excepción de la relacionada con el género (preferencia deshabilitada cuando existen peticiones, en pasos sucesivos se ofrece una explicación de por qué), junto con la información de **id**, **energía** y **bpm** de la petición.

```
var gatheringQuery = "MATCH (petition:Song)-[:WAS_ASKED]-(p:Person {user:'" + usuario + "'})-[:LAST_PLAYED]->(played:Song), " +  
"(p)-[:DESIRES]-(desire:Song) RETURN petition.id, toFloat(petition.bpm), toFloat(petition.energy), " +  
"played.id, toFloat(played.bpm), toFloat(desire.energy)";
```

Figura 30: recopilación de información en el paso 2.b

3. FILTRADO

a) NO EXISTEN PETICIONES

1. **Filtrado por género:** si se ha seleccionado la casilla de mantener género, se realiza una petición de todas las canciones enmarcadas en el mismo género que la reproducción actual. Si no se ha marcado la casilla, se pasa directamente al paso 3.a.2. Además, en este y todos los pasos sucesivos, se comprueba que los *tracks* devueltos no hayan sido reproducidos, gracias a la relación *PLAYED*. A partir de este paso, si se devuelven tres canciones o menos, finaliza el proceso.

```
/*Se filtra por genero (y por canciones que no hayan sido reproducidas)*/  
var genreQuery = "MATCH (s:Song), (p:Person {user: '" + usuario + "'}) WHERE s.genre='" + generoUltimo +  
" AND NOT (p)-[:PLAYED]->(s) RETURN s.id, s.title, s.artist, s.bpm, s.genre, s.cover, s.preview";
```

Figura 31: filtrado por género e historial

2. **Filtrado por bpm:** como se ha mencionado a lo largo de la memoria, el bpm es importante para el DJ, ya que la proximidad en *tempo* de dos canciones le permite mezclarlas de forma más fácil. Por eso, se filtran las canciones, además de por el criterio anterior de género (si fuese el caso) y por historial, por su bpm, con un margen de $\pm 5\%$ respecto al valor de bpm de la reproducción actual.

```
var bpmQuery = "MATCH (s:Song), (p:Person {user: '" + usuario + "'}) WHERE s.genre='" + generoUltimo +  
" AND NOT (p)-[:PLAYED]->(s) AND s.bpm >= " + (bpmUltimo - bpmUltimo*0.05) + " AND s.bpm <= " +  
" (bpmUltimo + bpmUltimo*0.05) + " RETURN s.id, s.title, s.artist, s.bpm, s.genre, s.cover, s.preview";
```

Figura 32: filtrado por bpm, género e historial

3. **Filtrado por energía:** si continúa habiendo más de 3 resultados, además de los criterios anteriores, se aplica un filtrado por energía, tomando el valor de energía recuperado del *nodo deseo*, y aplicando un margen de $\pm 10\%$.

```
var energyQuery = "MATCH (s:Song), (p:Person {user: '" + usuario + "'}) WHERE s.genre='" + generoUltimo +  
" AND NOT (p)-[:PLAYED]->(s) AND s.bpm >= " + (bpmUltimo - bpmUltimo*0.05) + " AND s.bpm <= " +  
" (bpmUltimo + bpmUltimo*0.05) + " AND s.energy >= " + (energiaPreferencia - 10) +  
" AND s.energy <= " + (energiaPreferencia + 10) + " RETURN s.id, s.title, s.artist, " +  
" s.bpm, s.genre, s.cover, s.preview";
```

Figura 33: filtrado por energía, bpm, género e historial



b) EXISTEN PETICIONES

En el caso de que existan peticiones, el proceso de filtrado se plantea de manera completamente distinta, calculando el camino mínimo entre el nodo de reproducción actual y la petición mediante el algoritmo de *Dijkstra*.

Recordemos que las peticiones son *particulares*, es decir, no suelen ser realizadas por el conjunto del público, si no por un miembro de éste, pidiéndola por tanto de acuerdo a su gusto particular. Es muy común que se pidan *tracks* alejados en energía y bpm de la canción que se está reproduciendo, e incluso en ocasiones sucede que el género de la petición es completamente distinto.

Por tanto, y retomando el concepto de sesión musical armoniosa descrito en el punto 5.1, se trata de lograr un cambio desde la reproducción actual (en adelante conocida como **origen**) a la petición (en adelante conocida como **destino**), tratando de que el público no perciba el cambio como demasiado brusco y sin relación alguna.

Y son precisamente las relaciones entre canciones en el *dataset* las que posibilitan este cambio. Una relación definía la *proximidad* de dos canciones entre sí, no solo teniendo en cuenta las características intrínsecas de la canción, si no además la posibilidad de que un usuario escuchase una después de la otra. Gracias a estas relaciones, podemos formar cientos de caminos por los que una persona llegaría de una a otra canción. Pero como el interés reside en reproducir la petición lo antes posible, se busca el camino mínimo.

Dijkstra permite calcular la distancia mínima entre dos nodos, origen y destino, analizando las distancias desde un determinado nodo hacia todos sus vecinos, eligiendo en cada iteración el camino más corto.

Como es probable que en este camino aparezcan nodos de géneros distintos al de origen y destino (sobre todo si el género del origen y el destino no coinciden), decae el sentido de mantener el género, como se mencionó en el paso 2.b.

En definitiva, se siguen los siguientes pasos de filtrado para una situación con peticiones. Al igual que ocurre en el caso 3.a, todos los pasos se siguen de forma acumulativa, finalizando en cualquier momento el proceso si se reciben 4 resultados o menos (el origen y el destino estarán incluidos en los resultados)

1. **Filtrado por Dijkstra:** se devuelven únicamente los nodos que forman parte del camino mínimo entre el origen y el destino, incluidos ambos. Es importante remarcar que este algoritmo se ejecuta gracias a la *Graph Data Science Library*, y requiere de la creación del grafo catalogado *grafoCanciones*, explicado en el punto 4.1.

```
var dijkstraQuery = "MATCH (source:Song {id:" + idUltimo + "}), (target:Song {id:" + idPetición + "}) " +  
"CALL gds.shortestPath.dijkstra.stream('grafoCanciones', {sourceNode: source, " +  
"targetNode: target}) YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path " +  
"WITH [nodeId IN nodeIds | gds.util.asNode(nodeId)] AS nodes UNWIND nodes as s WITH s " +  
"RETURN s.id, s.title, s.artist, s.bpm, s.genre, s.cover, s.preview";
```

Figura 34: filtrado por Dijkstra

2. **Filtrado por bpm:** de la misma forma que en el paso 3.a.2, se ejecuta el filtrado de los resultados por bpm, con la diferencia del margen aplicado. En esta ocasión el margen de bpm en el que se deben encontrar los resultados está comprendido entre el valor de bpm de la reproducción actual y el bpm de la petición, es decir, (bpmPetición, bpmReproducción) o (bpmReproducción, bpmPetición), dependiendo de cual de los dos valores sea mayor.

```
bpmQuery = "MATCH (source:Song {id:" + idUltimo + "}), (target:Song {id:" + idPetición + "}) " +  
"CALL gds.shortestPath.dijkstra.stream('grafoCanciones', {sourceNode: source, " +  
"targetNode: target}) YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path " +  
"WITH [nodeId IN nodeIds | gds.util.asNode(nodeId)] AS nodes UNWIND nodes as s WITH s " +  
"WHERE s.bpm >= " + (bpmPetición) + " AND s.bpm <= " + (bpmUltimo) +  
" RETURN s.id, s.title, s.artist, s.bpm, s.genre, s.cover, s.preview";
```

Figura 35: filtrado por Dijkstra y bpm,
cuando bpmPetición < bpmReproducción

3. **Filtrado por energía:** si continua habiendo más de 3 resultados, se filtran por el valor de energía, aplicando un criterio similar al del paso anterior: el margen de energía estará comprendido entre el valor de energía del *nodo deseo* y el de la petición, es decir, (energíaPetición, energíaDeseo) o (energíaDeseo, energíaPetición), según corresponda, aplicando además un margen de +-10% a cada valor.



```
energyQuery = "MATCH (source:Song {id:" + idUltimo + "}), (target:Song {id:" + idPetición + "}) " +  
  "CALL gds.shortestPath.dijkstra.stream('grafoCanciones', {sourceNode: source, " +  
  "targetNode: target}) YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path " +  
  "WITH [nodeId IN nodeIds | gds.util.asNode(nodeId)] AS nodes UNWIND nodes as s WITH s " +  
  "WHERE s.bpm >=" + (bpmPetición) + " AND s.bpm <=" + (bpmUltimo) + " AND s.energy >=" + (energiaPreferencia*0.90) +  
  " AND s.energy <=" + (energiaPetición*1.10) " RETURN s.id, s.title, s.artist, s.bpm, s.genre, s.cover, s.preview";
```

Figura 36: filtrado por Dijkstra, bpm y energía

4. ORDENACIÓN

Los resultados solo se ordenan si, recorridos todos los pasos de filtrado anteriores, aún se siguen devolviendo más de 3 resultados. Según el itinerario escogido (existencia o no de peticiones), los resultados se ordenan por puntuación en **PageRank personalizado**, o por distancia al nodo petición según **Dijkstra**.

a) HAY PETICIONES -> DIJKSTRA

Dijkstra, que ya fue aplicado en el paso anterior como método de filtrado, se utiliza ahora como forma de ordenar los resultados. Por defecto, en las peticiones mostradas en figuras anteriores, los resultados ya se ordenan según distancia al nodo destino, por lo que no es necesario código adicional, más que para encapsular la respuesta en un array de objetos JavaScript.

b) NO HAY PETICIONES -> PAGERANK PERSONALIZADO

PageRank mide la importancia de un determinado nodo dentro de un grafo, teniendo en cuenta dos factores: la cantidad de relaciones que apuntan hacia él, y la importancia de los nodos a los que está conectado.

Por ejemplo, en el dataset de LIVE, una canción x sumamente popular será aquella relacionada con muchas otras. Por otro lado, si una canción y de nicho (poco conocida) tiene la suerte de estar relacionada con x, es muy probable que y cuente con un PageRank mayor que otras vecinas, que, aún teniendo más relaciones que y, todas ellas son con canciones desconocidas para el público general.

Al ordenar los resultados, podría haberse optado por otra solución, como extraer únicamente los vecinos directos de



la reproducción actual, que aparentemente, contarían con una mayor *proximidad*. Sin embargo, la proximidad no es garantía de *valor*.

Introducido en el paper original de Google en 1998, *PageRank* ofreció al buscador de la G una ventaja competitiva, al ofrecer resultados de una mayor relevancia o *valor* que los de sus competidores, los cuales se limitaban a hacer un filtrado en base a palabras clave o etiquetas. Su fórmula es la siguiente:

$$PR(A) = (1 - d) + d\left(\frac{PR(T_1)}{C(T_1)} + \dots + \frac{PR(T_n)}{C(T_n)}\right)$$

Donde A es un determinado nodo de la red, $T_1 \dots T_n$ los distintos nodos con los que A está conectado, $C(T_n)$ el número de relaciones que salen del nodo T_n y d , el factor de amortiguamiento, que en LIVE es igual a 0.85.

PageRank se repite hasta un determinado número de iteraciones, o bien hasta que la variación de los valores obtenidos sea mínima. En el caso de LIVE, se configuran 20 iteraciones.

Los valores de *PageRank* varían en función del nodo por el que se empieza a ejecutar el algoritmo. Originalmente, se planteaba que este nodo fuese elegido al azar. Sin embargo, con la llegada de la Web 2.0 y las RRSS a finales de los 2000, se popularizó una variante conocida como *PageRank personalizado*, en la cual se puede definir el nodo inicio, representando así la relevancia respecto al nodo origen. Ésta es la variante empleada en este proyecto, tomando la reproducción actual como el nodo origen.

Al igual que al ejecutar *Dijkstra*, se utiliza el grafo catalogado *grafoCanciones* para aumentar el rendimiento, por lo que conviene revisar si no ha sido eliminado por un reinicio de la BBDD.



```
var pageRankQuery = "MATCH (s:Song {id:" + idUltimo +
    "}) CALL gds.pageRank.stream('grafoCanciones', { maxIterations: 20," +
    " dampingFactor: 0.85, sourceNodes: [s] }) YIELD nodeId, score" +
    " WITH gds.util.asNode(nodeId) as s, score MATCH (p:Person {user: '" + usuario +
    "''}) WHERE s.genre='" + generoUltimo + "' AND NOT (p)-[:PLAYED]->(s) " +
    "AND s.bpm >=" + (bpmUltimo - bpmUltimo*0.05) + " AND s.bpm <=" +
    (bpmUltimo + bpmUltimo*0.05) + " AND s.energy>=" + (energiaPreferencia - 10) +
    " AND s.energy<=" + (energiaPreferencia + 10) + " RETURN s.id, s.title, s.artist, " +
    "s.bpm, s.genre, s.cover, s.preview ORDER BY score DESC LIMIT 3";
```

Figura 37: filtrado mediante energía, bpm, género, historial y ordenación de los resultados mediante valor en PageRank



6. EJECUCIÓN Y ANÁLISIS DE RESULTADOS

A continuación se presentan diferentes ejemplos de uso de LIVE, como demostración de sus capacidades de recomendación y aplicación en diferentes situaciones.

6.1. MISMO TRACK, MISMA HORA, DOS SESIONES

Como se ha recalcado a lo largo de esta memoria, la principal diferencia de LIVE respecto a otro sistema recomendador de música reside en valorar el contexto, emitiendo recomendaciones distintas para un mismo *track*, si el contexto del momento así lo requiere. La principal característica que implementa este principio es el modo automático de energía, que calcula valores de intensidad musical distintos dependiendo del momento de la sesión.

En el siguiente caso, se desea obtener recomendaciones para el track *Wake Me Up* del DJ sueco *Avicii*. Dicha pista de música electrónica de baile (EDM) tiene un valor de energía de 78.3% en el *dataset* de LIVE, y, si bien puede considerarse una canción eufórica, triunfando en todos los festivales en la primera mitad de los 2010, contiene elementos que podrían tildarse de melancólicos, gracias a su envoltura *folk* y la voz de *Aloe Blacc*.

Se configurarán dos sesiones distintas, situando la hora a la que se redacta este caso de uso en el principio y la mitad de una sesión, respectivamente.

6.1.1. CASO 1: 01:30 A 06:30

Con la ya mencionada canción seleccionada, se inicia sesión y se configura una sesión de 5 horas de duración, con inicio a la 01:30 y final a las 06:30. Se observa que según el modo automático se obtiene un 62% de energía, al calcular la media entre el valor de energía del *track* y el valor de energía ideal para esa hora (cercano al 30% del momento inicial de la sesión).

Sin marcar la casilla de mantener género, se obtienen 3 recomendaciones. Ninguna es de una canción de EDM, dado que éstas suelen tener un valor de energía más alto. Las tres canciones están en la horquilla de los 122 y los 128 BPM, cumpliendo con el margen de $\pm 5\%$ respecto a los 124



BPM de *Wake Me Up*. Los tres tracks se enmarcan en la primera mitad de los 2010, al igual que la reproducción original. Son *Counting Stars*, de la banda *One Republic* un tema *pop-folk* con una atmósfera bastante híbrida, como el sencillo de *Avicii*; *Habbits (Stay High) (Hippie Sabbotage Remix)*, de *Tove Lo*, un track *downtempo* con una atmósfera melancólica, pero una línea de bajo muy marcada que lo hace perfecto para el sistema de sonido de un local de ocio nocturno; y *Somebody That I Used To Know*, del músico australiano *Gotye*, *indie-pop/folk* pero no muy apto para el baile.

Se concluye así, que los tres tracks aplican una rebaja a la energía para no *quemarla* demasiado pronto, guardan una relación de época, contienen trazas de la canción original, un tempo similar, y dos de tres pueden considerarse aptos para la pista de baile.

Avanzando ligeramente en la noche, y manteniendo el track de *Avicii*, a las 02:13 se recargan las recomendaciones, esta vez manteniendo el género. El valor de energía estimado por LIVE se sitúa en el 68%.

Se obtienen 3 canciones: *Waiting for Love*, del mismo *Avicii*; *This Girl*, de *Kungs*; y *Sun Goes Down*, de *Robin Schulz*. Las tres canciones se lanzaron en la primera mitad de los 2010, y podrían enmarcarse en la corriente *tropical house*, subgénero del EDM que destaca por una atmósfera más relajada, un ambiente más hedonista y veraniego, con melodías pegadizas y bailables. El tempo de estos *tracks* se sitúa entre los 121 y los 127 BPM.

En definitiva, los tres tracks mantienen el género, cumplen con el margen de *tempo*, y ofrecen soluciones más enérgicas que hace unos minutos, aún tranquilas y a la vez bailables.

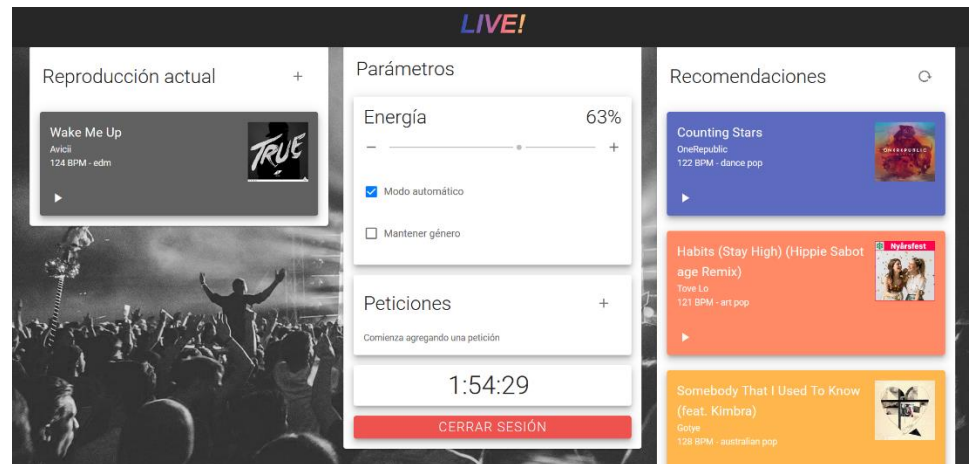


Figura 38: caso 1.1 sin mantener el género

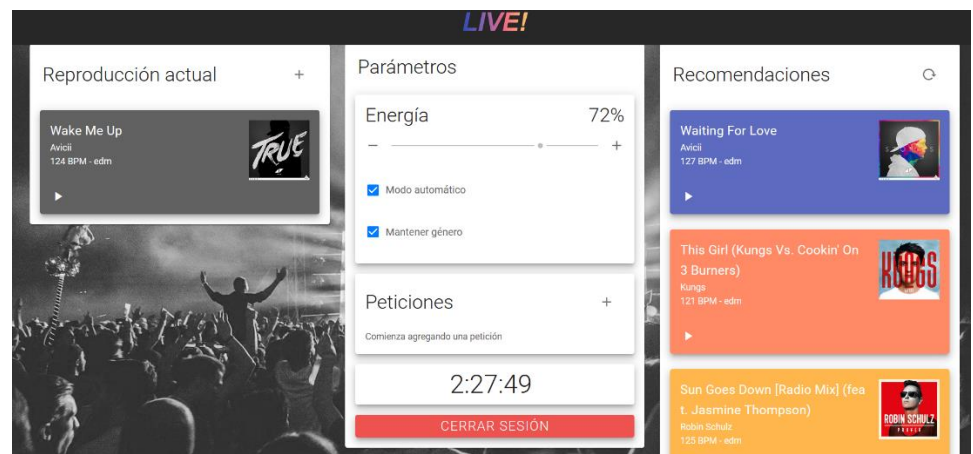


Figura 39: caso 1.1 manteniendo el género (captura realizada posteriormente)

6.1.2. CASO 2: 23:30 a 05:30

Son las 02:39 y se configura una nueva sesión, con inicio en las 23:30 y fin a las 05:30. De esta manera, el valor de energía ideal estará próximo a su valor máximo del 90%. El valor de energía calculado por LIVE se sitúa en el 84%.

Sin mantener el género, se obtienen tres canciones, dos de ellas EDM y una rock. Son *Summer*, del DJ *Calvin Harris*, tema muy energético y ritmo muy marcado, con un 85.6% de energía según el *dataset* de LIVE; *Sugar*, del ya mencionado *Robin Schulz*, aún *tropical house*, pero más intenso; y *Believer*, de la banda *Imagine Dragons*, clasificado como



arena rock, cargado de instrumentación y marcados estribillos.

Los tracks abarcan entre los 123 y los 128 bpm. Puede asegurarse así que cumplen con el margen de *tempo*, ofrecen soluciones más enérgicas que la reproducción actual, e incluso, la posibilidad de variar de género.

Si se desea mantener el género y se recargan las recomendaciones, solo cambia uno de los tracks: desaparece *Believer* y entra *Blame*, de *Calvin Harris*, con 128 bpm, un valor de energía de 85.7%, y las vocales de *John Newman*, las cuales *casan* con el registro soul del cantante de *Wake Me Up*.

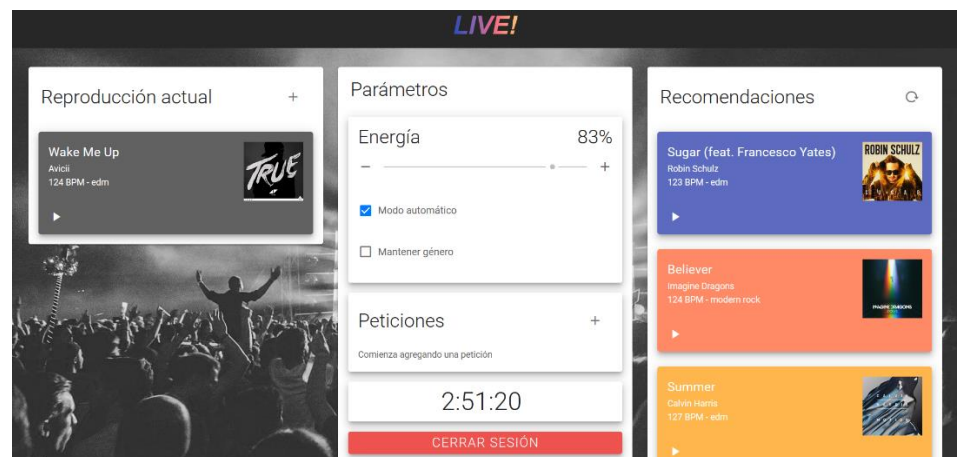


Figura 40: caso 1.2, sin mantener el género

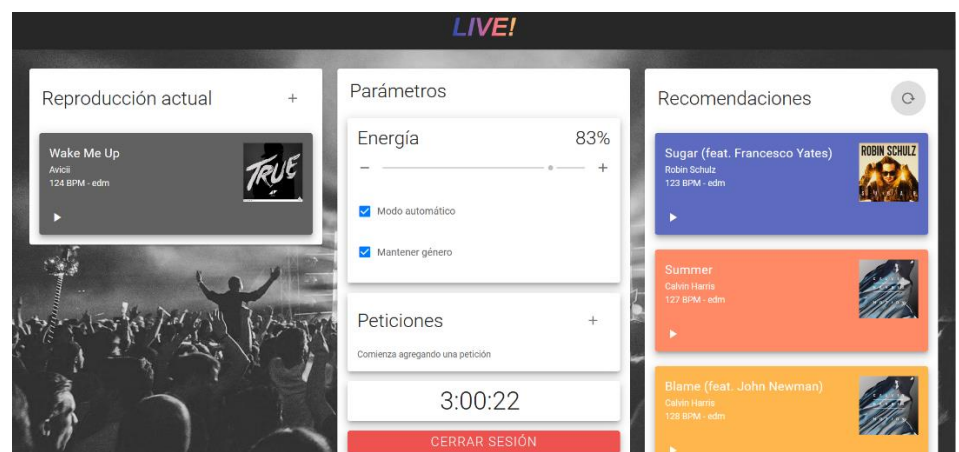


Figura 41: caso 1.2, manteniendo el género

6.2. HOY NO TOCA REPETIR

En esta versión inicial, LIVE almacena el historial de reproducciones del usuario hasta que éste cierra sesión, de manera que no se emitan recomendaciones de canciones ya reproducidas. Esto no evita, por supuesto, que el DJ decida libremente repetir un *track*, en cuyo caso seguirá pudiendo seleccionarlo en el buscador.

Vamos a poner que sucede esto último. En primer lugar, se decide reproducir el track *Limbo*, de *Daddy Yankee*. Después, se obtienen recomendaciones manteniendo el género, en este caso el reggaetón.

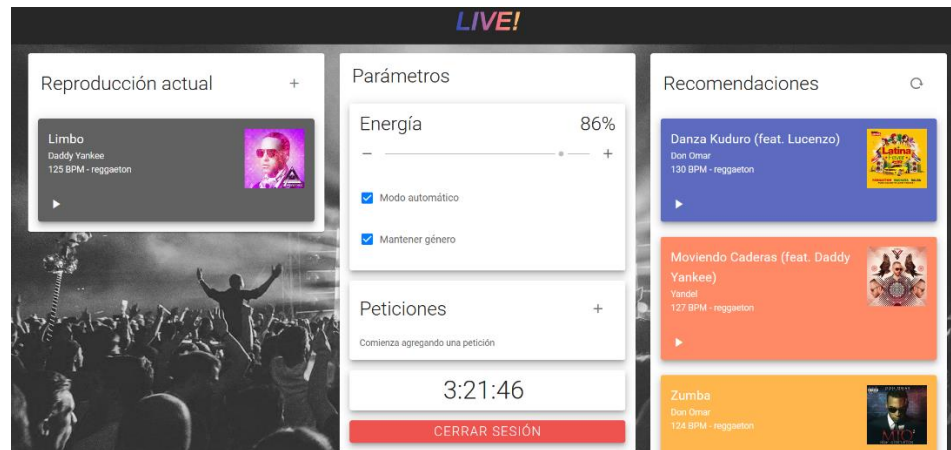


Figura 42: caso 2, tras obtener recomendaciones

Se obtienen tres recomendaciones: la *Danza Kuduro*, de *Don Omar*; *Moviendo Caderas*, de *Yandel* y el mismo *Daddy Yankee*; y *Zumba* de *Don Omar*. Pongamos que decidimos reproducir la *Danza Kuduro*, para lo cual la seleccionamos en el buscador ofrecido por el componente de reproducción actual.

Si recargamos las recomendaciones, como es lógico, desaparece la *Danza Kuduro* y entra *Adrenalina*, de *Wisin*. A continuación, seleccionemos *Moviendo Caderas*.

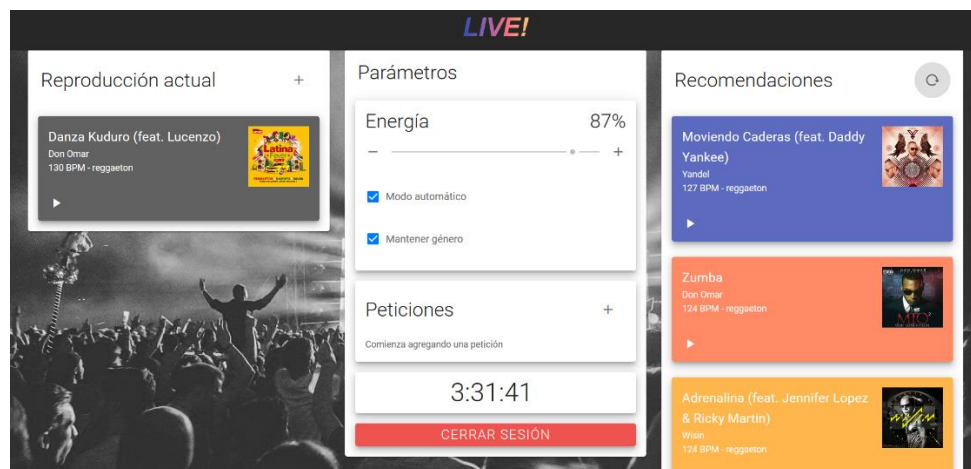


Figura 43: caso 2, tras seleccionar la Danza Kuduro

Cambian las tres recomendaciones. Pero recordemos que de todas las que obtuvimos originalmente, al reproducir *Limbo*, solo queda por seleccionar una: *Zumba*. Por un momento, y con el mero fin de evaluar el funcionamiento de LIVE, vamos a obviar las recomendaciones.

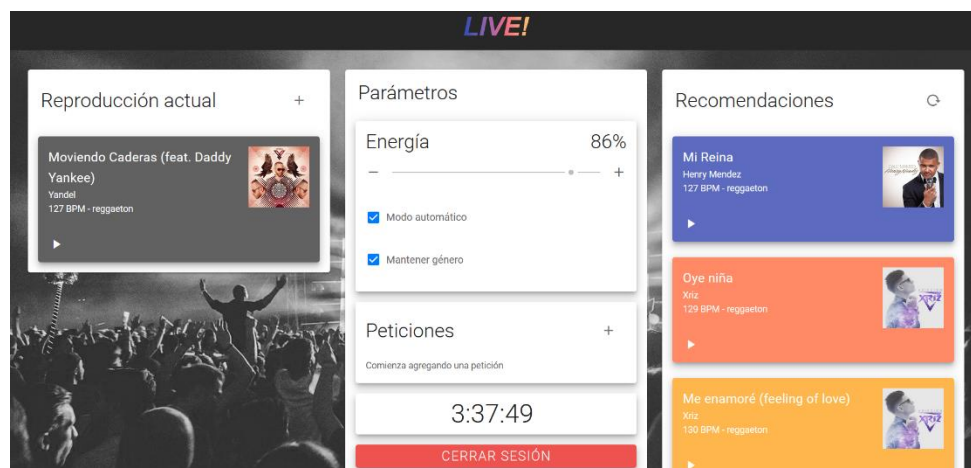


Figura 44: caso 2, tras seleccionar Moviendo Caderas

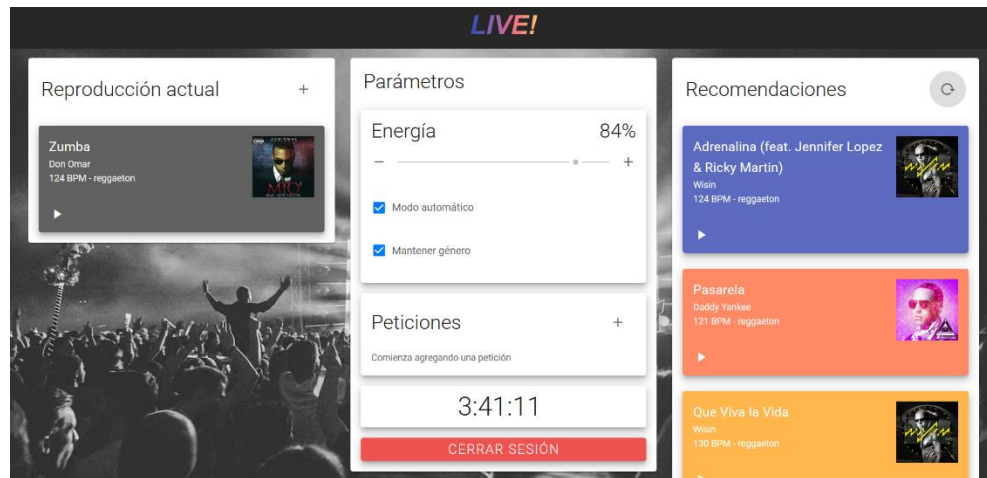


Figura 45: caso 2 tras seleccionar Zumba

Finalmente, si volvemos a seleccionar Limbo, porque así lo deseamos, encontraremos tres recomendaciones nuevas, de reggaetón, en los márgenes de *tempo*, y garantizando, como ya se enunció al principio de esta memoria, que no haya dos momentos iguales, de la misma forma que tampoco hay dos sesiones iguales.

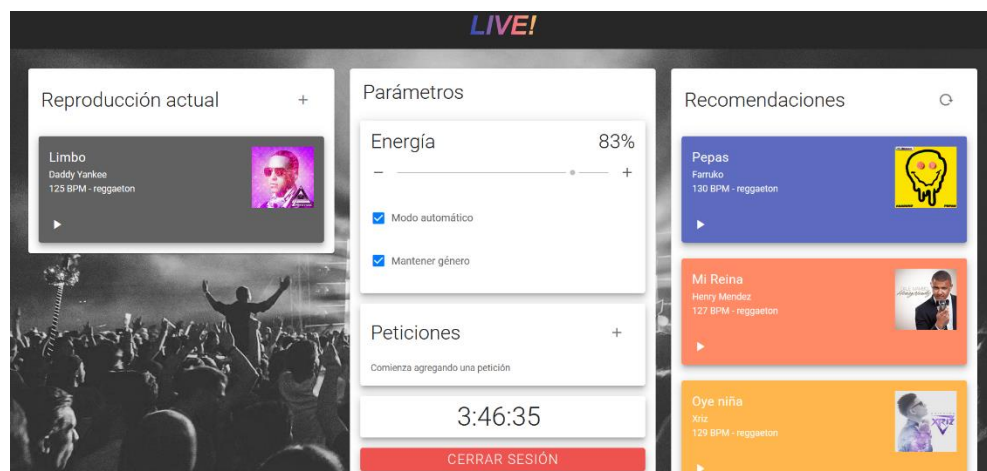


Figura 46: caso 2, sin repetición tras volver a seleccionar Limbo

6.3. DE GÉNERO EN GÉNERO, DEL ORIGEN A LA PETICIÓN

Como último caso de uso en la que la recomendación contextual de LIVE juega un papel clave, se analizará un caso en el que se introduce una petición en el sistema. Manteniendo *Limbo* de *Daddy Yankee*, un *track* que oscila entre el reggaetón y el electro latino, uno de los miembros del público solicita, *Caminando por la Vida*, sencillo de *Melendi* encuadrado en el flamenco y la rumba.

Tal y como se describió en el punto 5.2, cuando existen peticiones, LIVE utiliza el algoritmo de caminos mínimos de *Dijkstra* para tratar de *alcanzar* la petición de la forma más rápida y armoniosa posible, utilizando las relaciones de tipo *RELATED*. Ejecutando este caso de uso, se obtienen los siguientes resultados.

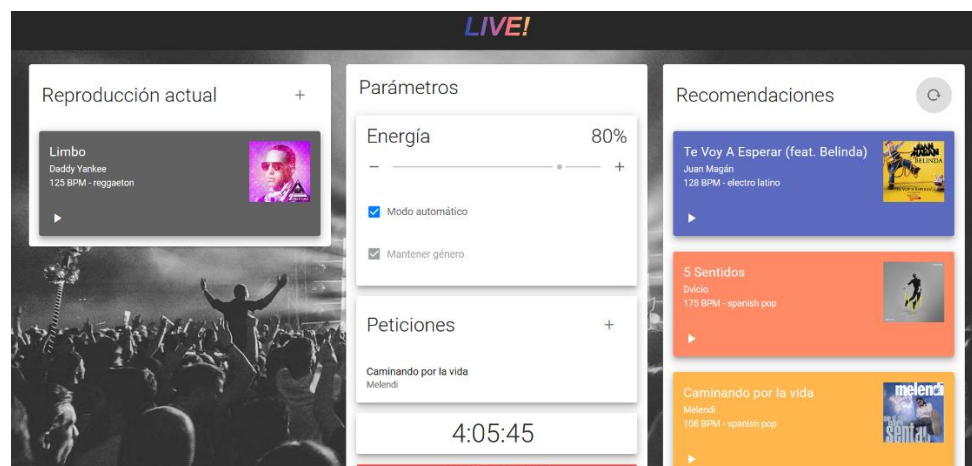


Figura 47: resultados al ejecutar el caso 3

Se observan tres resultados en la figura superior. Si se ejecuta la misma petición de camino mínimo en neo4j Browser, comprobamos que no se ha necesitado hacer ningún filtrado por bpm o energía, ya que el camino de *Limbo* a *Caminando por la vida* está formado por cuatro nodos, incluidos el origen y el destino.

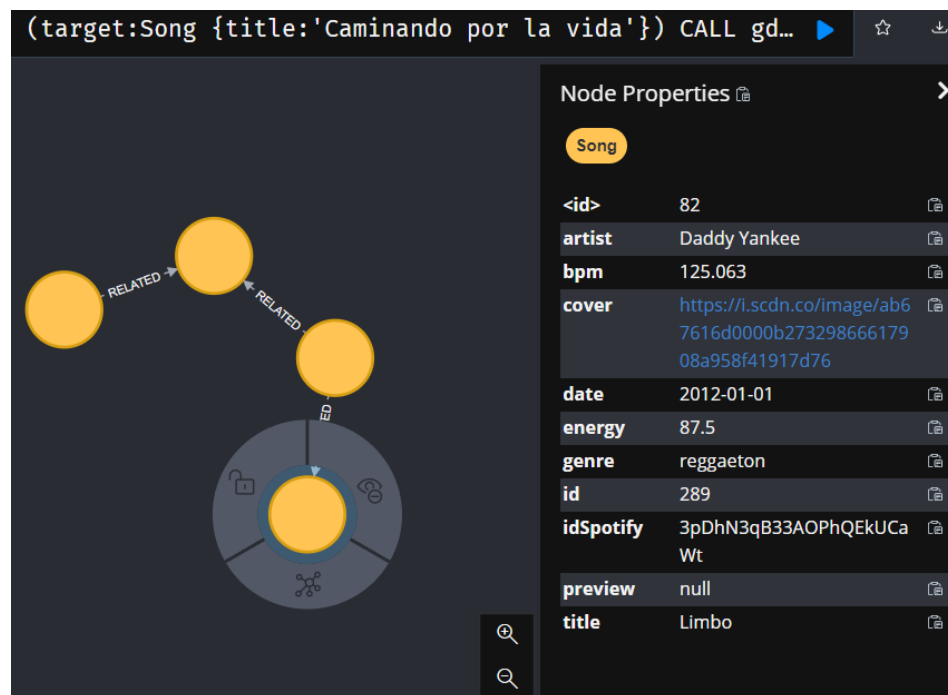


Figura 48: camino mínimo en neo4j Browser

Analizando la calidad de los resultados, podemos determinar patrón común entre todas las canciones: una letra fácil de cantar a coro, sobre todo cuando cruzamos la línea del pop español. En primer lugar, se ofrece la canción *Te Voy a Esperar* de Juan Magán, que abandona los tintes de reggaetón de *Limbo*, empezando a coquetear con sonidos más *pop*, finalizando la transición con *5 Sentidos*, un sencillo fácil de entonar, hasta la canción destino.

Se plantea así una experiencia donde los sonidos más propios de sintetizador van dejando paso progresivamente a una instrumentación más convencional, menos electrónica. De género en género, del origen a la petición.



7. ANÁLISIS DAFO

En el siguiente apartado se realiza un análisis DAFO (Debilidades, Amenazas, Fortalezas y Oportunidades) del proyecto desarrollado:

- **Debilidades:** en cuanto a los factores internos que restan ventaja a LIVE, destacaría la dependencia de datos de terceros, sobre todo para determinar la energía de las canciones. La ausencia de un sistema de análisis musical propio hace que en ocasiones se haya sentido que la valoración de energía ofrecida en el *daataset* no es completamente representativa, algo que trata de solucionarse mediante la aplicación de márgenes de filtrado con posterioridad al cálculo de la energía estimada para un determinado momento.
Por otro lado, la propia naturaleza de la aplicación, planteada exclusivamente como un sistema de recomendación y apoyo, la expone a una desventaja: LIVE no es un DJ. LIVE sugiere qué mezclar, pero no ofrece las herramientas para hacerlo.
- **Amenazas:** en línea con lo mencionado en el último párrafo, cabe destacar que las tecnologías implementadas por LIVE están al alcance de la mano de cualquier otra organización, lo que hace que una *suite* de DJ o incluso un servicio de *streaming* pudiesen integrar sus características como una funcionalidad más de sus respectivos productos.
- **Fortalezas:** es imposible no mencionar la interfaz de usuario, que ofrece una experiencia muy limpia y atractiva visualmente, a la par de fácil. La utilización de una base de datos basada en grafos ofrece también un rendimiento excelente, además de una gran escalabilidad, tanto en datos como en usuarios.
- **Oportunidades:** el *dataset* de ejemplo generado, aún con sus puntos débiles mencionados en el primer apartado de esta sección, fue construido teniendo en mente la música que suena en los locales de ocio de España, lo que lo hace prácticamente único en el mercado. Así mismo, la vocación no generalista y de nicho de LIVE hace improbable en el corto y medio plazo la proliferación de varias alternativas.

8. EVOLUCIÓN PREVISIBLE DEL SISTEMA

Con los datos presentes a lo largo de la ejecución de LIVE, y más concretamente con el historial de reproducción, es evidente que éste podría ser utilizado para algo más que evitar canciones que ya hayan sido reproducidas. Ya que el fin principal de LIVE es ofrecer una recomendación lo más basada en contexto posible, podría utilizarse el historial para obtener una serie de métricas que permitan aproximarse más a la realidad de este contexto, como bien podría ser calcular una media de energía de todos los *tracks* hasta la fecha, permitiendo conocer más la naturaleza, extroversión o *desenfreno* del público.

Otro dato interesante a observar en el público es su edad. Es cierto que hay *canciones que no mueren*, pero también es cierto que hay géneros o artistas con los cuales una generación tiende a identificarse más que otra. No es casualidad que varias de las 10 canciones descritas en el apartado 3.1 hayan sido publicadas durante la década de los 2000 y primeros de los 2010.

Que estas canciones sean omnipresentes en casi todas las discotecas de León no es casualidad: diversos estudios demuestran que los individuos tienden a identificarse más con las canciones de su niñez tardía y adolescencia. Según uno de ellos, elaborado a partir de datos de Spotify y del que se hizo eco *The New York Times*, las canciones más influyentes para una persona tendían a ser las que había escuchado a los 13 años, en el caso de las mujeres, y a los 14, en el caso de los hombres. El público objetivo mayoritario de los locales de ocio nocturno son los universitarios, los cuales son ahora mismo nacidos entre finales de los 90 y primeros 2000, lo que confirma la selección musical.

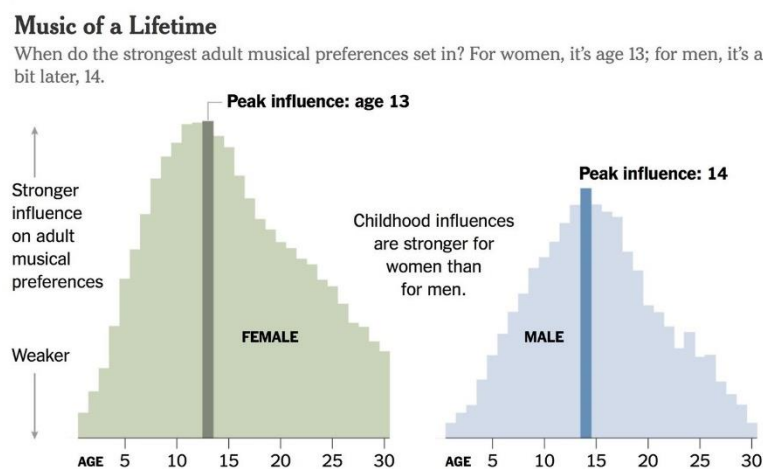


Figura 49: Music of a Lifetime (Fuente: NYTimes)



9. LECCIONES APRENDIDAS

Prácticamente toda la elaboración de este proyecto ha supuesto para mí un aprendizaje. La primera lección sin duda son los conocimientos adquiridos sobre **neo4j**. Hasta el momento, solo había trabajado con bases de datos relacionales, y en mi opinión, las bases de datos basadas en grafos ofrecen una aproximación certera a los problemas de la vida real, los cuales se basan en la interacción de entidades o individuos. Destacaría además la versatilidad que ofrece **Cypher** y el poder olvidarse de las muchas veces complejas Foreign Keys de SQL.

Una vez me sentí con la confianza y el conocimiento suficientes de neo4j, para lo cual realicé cinco cursos ofrecidos en su plataforma GraphAcademy, me lancé a abordar los sistemas de recomendación, conociendo las herramientas ofrecidas por la propia plataforma de bases de datos. Desde que planteé el problema tuve claro las variables a observar, sin embargo, desconocía como *atarlas*, sacarles el máximo partido. Las herramientas ofrecidas por la **Graph Data Science Library**, me han permitido tener un primer acercamiento a la ciencia de datos bastante divertido, y quedarme con ganas de profundizar e incluso orientar mi futuro profesional hacia esta rama.

Por supuesto, no creo que me hubiese divertido tanto con esta experiencia de no ser por la temática que la rodea: la **música**. Me ha resultado bastante interesante conocer determinadas relaciones entre algunas canciones, y he hecho algún que otro descubrimiento musical. Otro reto que me propongo ahora que la aplicación está finalizada es ponerla en uso, para lo cual, tendré que ampliar mis conocimientos sobre mezclar música, ya que como menciono en el DAFO, LIVE no es un DJ.

Cerrada la parte de los datos, afronté lo que más miedo me daba, la parte en la que vi peligrar más el trabajo: **programar una aplicación web**. Aún con más que suficiente experiencia de programación a mis espaldas, nunca había trabajado con el modelo de diseño back-front, ni tampoco consideraba haber hecho una interfaz de usuario de la que me sintiera lo suficiente orgulloso. El *framework* con más sofisticación que había utilizado era Java Swing, con interfaces más propias de la década de los 2000 que de hoy en día.

Vue.js y **Vuetify** ofrecen una gran cantidad de componentes y un modelo de programación orientada eventos que me fue muy fácil manejar. Aparte, **express** y **node.js** me han abierto al mundo de las API REST y el manejo de peticiones HTTP.



Como resumen, a nivel personal este proyecto me ha aportado confianza en mi mismo, al haber seguido un camino prácticamente autodidacta, fruto de cursar esta asignatura un año antes del curso en el que está enmarcada (3º en vez de 4º), y sin la asignatura donde se enseñan muchas de las tecnologías utilizadas (Aplicaciones Web). Tampoco podría haber seguido este camino sin la disposición mostrada en todo momento por el profesor, Enrique, quién nos proporcionó a todos los alumnos de la asignatura los recursos, la atención personalizada, y sobre todo, las sonrisas que nos sacó en algún que otro momento del semestre.



10. BIBLIOGRAFIA, REFERENCIAS Y RECURSOS

- ytmusicapi: Unofficial API for YouTube Music — ytmusicapi 0.20.0 documentation. (2018). Ytmusicapi Documentation. <https://ytmusicapi.readthedocs.io/en/latest/index.html>
- Welcome to Spotipy! — spotipy 2.0 documentation. (2021). Spotipy 2.0 Documentation. <https://spotipy.readthedocs.io/en/2.19.0/>
- Introduction a Vue.js. (2021). Vue.Js. <https://vuejs.org/v2/guide/>
- Bluuweb. (2018, 7 noviembre). Curso de Vue.js #01 Introducción [Tutorial en Español desde cero] Framework de Javascript [Vídeo]. YouTube. <https://www.youtube.com/watch?v=GAQB7Y4X5fM&list=PLPl81lqbj-4J-gfAERGDCdOQtVgRhSvIT&index=1>
- neo4j. (2021). GraphAcademy. Neo4j Graph Database Platform. <https://neo4j.com/graphacademy/>
- IFPI. (2021, octubre). Engaging with Music 2021. <https://www.ifpi.org/wp-content/uploads/2021/10/IFPI-Engaging-with-Music-report.pdf>
- Stephens-Davidowitz, S. (2018, 10 febrero). Opinion | The Songs That Bind. The New York Times. <https://www.nytimes.com/2018/02/10/opinion/sunday/favorite-songs.html>
- Brin, Page, S. L. (1998). The Anatomy of a Search Engine. Stanford University. <http://infolab.stanford.edu/%7Ebackrub/google.html>